

Curs 1

2021-2022

Fundamentele Limbajelor de Programare

Cuprins

- 1 Organizare**
- 2 Privire de ansamblu**
 - Bazele programării funcționale / logice
 - Semantica Limbajelor de Programare
- 3 Programare logică & Prolog**

Organizare

Instructori

Curs

- Ioana Leuștean (seriile 23, 25)**
- Denisa Diaconescu (seria 24)**

Laborator

- | | |
|----------|--|
| Seria 23 | <input type="checkbox"/> Ana Iova (Turlea) (231) |
| | <input type="checkbox"/> Horatiu Cheval (232) |
| | <input type="checkbox"/> Bogdan Macovei (233,234) |
| Seria 24 | <input type="checkbox"/> Natalia Ozunu (241) |
| | <input type="checkbox"/> Bogdan Macovei (242,243,244) |
| Seria 25 | <input type="checkbox"/> Ana Iova (Turlea) (251) |
| | <input type="checkbox"/> Bogdan Macovei (252) |

Resurse

- Seriile 23, 25
 - Moodle
 - Pagina externa
- Seria 24
 - Moodle
 - Pagina externa
- Suporturile de curs si laborator/seminar, resurse electronice
- Stiri legate de curs vor fi posteate pe Moodle si pe paginile externe

Prezenta

Prezenta la curs sau la laboratoare/seminarii nu este obligatorie, dar extrem de incurajata.

Notare



Notare

- Nota finală: 1 punct (oficiu) + examen

Notare

- Nota finală: 1 punct (oficiu) + examen
- Condiție minima pentru promovare: nota finală > 4.99

Notare

- Nota finală: 1 punct (oficiu) + examen
- Condiție minima pentru promovare: nota finală > 4.99
- Puncte bonus
 - La sugestia profesorului coordonator al laboratorului/seminarului, se poate nota activitatea în plus față de cerințele obisnuite
 - Maxim 1 punct

Examen

- În sesiunea
- Durată 2 ore
- Cu materialele ajutătoare
- Mai multe detalii vor fi oferite până la jumătatea semestrului

Curs/laborator

Curs

Bazele programării logice

- Logica clauzelor Horn, Unificare, Rezolutie

Semantica limbajelor de programare

- Semantică operațională, statică și axiomatică
- Inferarea automată a tipurilor

Bazele programării funcționale

- Lambda Calcul, Codificări Church, corespondența Curry-Howard
- Lambda Calcul cu tipuri de date

Curs/laborator

Curs

Bazele programării logice

- Logica clauzelor Horn, Unificare, Rezolutie

Semantica limbajelor de programare

- Semantică operațională, statică și axiomatică
- Inferarea automată a tipurilor

Bazele programării funcționale

- Lambda Calcul, Codificări Church, corespondența Curry-Howard
- Lambda Calcul cu tipuri de date

Laborator/Seminar:

Prolog Cel mai cunoscut limbaj de programare logică

- Verificator pentru un mini-limbaj imperativ
- Inferență tipurilor pentru un mini-limbaj funcțional

Haskell Limbaj pur de programare funcțională

- Interpretoare pentru mini-limbi

Exercitii suport pentru curs

Bibliografie

- B.C. Pierce, **Types and programming languages.** MIT Press.2002
- G. Winskel, **The formal semantics of programming languages.** MIT Press. 1993
- H. Barendregt, E. Barendsen, **Introduction to Lambda Calculus,** 2000.
- J. Lloyd. **Foundations of Logic Programming**, second edition. Springer, 1987.
- P. Blackburn, J. Bos, and K. Striegnitz, **Learn Prolog Now!** (Texts in Computing, Vol. 7),College Publications, 2006
- M. Huth, M. Ryan, **Logic in Computer Science (Modelling and Reasoning about Systems)**, Cambridge University Press, 2004.

Privire de ansamblu

Bazele programării funcționale / logice

Principalele paradigmă de programare

- Imperativă (cum calculăm)

- Procedurală
 - Orientată pe obiecte

- Declarativă (ce calculăm)

- Logică
 - Funcțională

Principalele paradigmă de programare

- Imperativă (cum calculăm)

- Procedurală
 - Orientată pe obiecte

- Declarativă (ce calculăm)

- Logică
 - Funcțională

Fundamentele paradigmelor de programare

Imperativă Execuția unei Mașini Turing

Logică Rezoluția în logica clauzelor Horn

Funcțională Beta-reductie în Lambda Calcul

Programare declarativă

- Programatorul spune **ce** vrea să calculeze, dar nu specifică concret **cum** calculează.
- Este treaba interpreterului (compilator/implementare) să identifice cum să efectueze calculul respectiv.
- Tipuri de programare declarativă:
 - Programare funcțională (e.g., Haskell)
 - Programare logică (e.g., Prolog)
 - Limbaje de interogare (e.g., SQL)

Programare funcțională

Esență: funcții care relaționează intrările cu ieșirile

- Caracteristici:**
 - funcții de ordin înalt – funcții parametrizate de funcții
 - grad înalt de abstractizare (e.g., functori, monade)
 - grad înalt de reutilizarea codului — polimorfism
- Fundamente:**
 - Teoria funcțiilor recursive
 - Lambda-calcul ca model de computabilitate
(echivalent cu mașina Turing)
- Inspiratie:**
 - Inferența tipurilor pentru templates/generics in POO
 - Model pentru programarea distribuită/bazată pe evenimente (callbacks)

Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.

Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.

- Unul din sloganurile programării logice:

Program = Logică + Control (R. Kowalski)

Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.
- Unul din sloganurile programării logice:
Program = Logică + Control (R. Kowalski)
- Programarea logică poate fi privită ca o deductie controlată.

Programare logică

□ Programarea logică este o paradigmă de programare bazată pe logică formală.

□ Unul din sloganurile programării logice:

Program = Logică + Control (R. Kowalski)

□ Programarea logică poate fi privită ca o deductie controlată.

□ Un **program** scris într-un limbaj de programare logică este

o listă de formule într-o logică

ce exprimă fapte și reguli despre o problemă.

Programare logică

□ Programarea logică este o paradigmă de programare bazată pe logică formală.

□ Unul din sloganurile programării logice:

Program = Logică + Control (R. Kowalski)

□ Programarea logică poate fi privită ca o deductie controlată.

□ Un **program** scris într-un limbaj de programare logică este
o listă de formule într-o logică
ce exprimă fapte și reguli despre o problemă.

□ Exemple de limbi de programare logică:

- Prolog
- Answer set programming (ASP)
- Datalog

Semantica Limbajelor de Programare

Ce definește un limbaj de programare?

Sintaxa Simboluri de operație, cuvinte cheie, descriere (formală) a programelor/expresiilor bine formate

Practica Un limbaj e definit de modul cum poate fi folosit

- Manual de utilizare și exemple de bune practici
- Implementare (compilator/interpreter)
- Instrumente ajutătoare (analizor de sintaxă, verificator de tipuri, depanator)

Ce definește un limbaj de programare?

Sintaxa Simboluri de operație, cuvinte cheie, descriere (formală) a programelor/expresiilor bine formate

Practica Un limbaj e definit de modul cum poate fi folosit

- Manual de utilizare și exemple de bune practici
- Implementare (compilator/interpreter)
- Instrumente ajutătoare (analizor de sintaxă, verificator de tipuri, depanator)

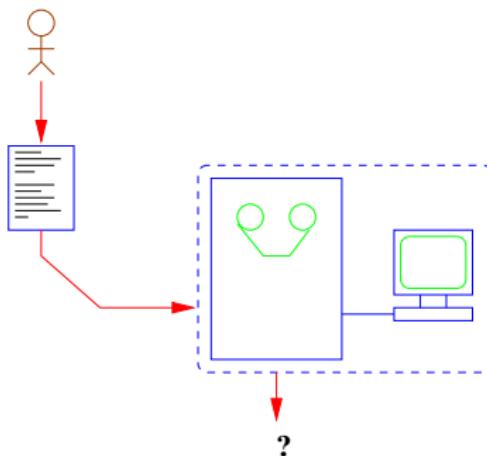
Semantica? Ce înseamnă / care e comportamentul unei instrucțiuni?

- De cele mai multe ori se dă din umeri și se spune că **Practica** e suficientă
- Limbajele mai utilizate sunt **standardizate**

La ce folosește semantica

- Să înțelegem un limbaj în profunzime
 - Ca programator: pe ce mă pot baza când programez în limbajul dat
 - Ca implementator al limbajului: ce garanții trebuie să ofer
- Ca instrument în proiectarea unui nou limbaj / a unei extensii
 - Înțelegerea componentelor și a relațiilor dintre ele
 - Exprimarea (și motivarea) deciziilor de proiectare
 - Demonstrarea unor proprietăți generice ale limbajului
E.g., execuția nu se va bloca pentru programe care trec de analiza tipurilor
- Ca bază pentru demonstrarea corectitudinii programelor.

Problema corectitudinii programelor



- Pentru anumite metode de programare (e.g., **imperativă, orientată pe obiecte**), nu este ușor să stabilim că un program este **corect** sau să înțelegem ce înseamnă că este corect (e.g., în raport cu ce?!).
- **Corectitudinea programelor** devine o problemă din ce în ce mai importantă, nu doar pentru aplicații "safety-critical".
- Avem nevoie de metode ce asigură "calitate", capabile să ofere "garanții".

C

```
#include <iostream>
using namespace std;
int main()
{
    int square;
    for(int i = 1; i <= 5; ++i)
    {
        square = i * i;
        cout << square << endl;
    }
}
```

C

```
#include <iostream>
using namespace std;
int main()
{
    int square;
    for(int i = 1; i <= 5; ++i)
    {
        square = i * i;
        cout << square << endl;
    }
}
```

Este corect?

C

```
#include <iostream>
using namespace std;
int main()
{
    int square;
    for(int i = 1; i <= 5; ++i)
    {
        square = i * i;
        cout << square << endl;
    }
}
```

Este corect? În raport cu ce?

```
#include <iostream>
using namespace std;
int main()
{
    int square;
    for(int i = 1; i <= 5; ++i)
    {
        square = i * i;
        cout << square << endl;
    }
}
```

Este corect? În raport cu ce?

Un formalism adecvat trebuie:

- să permită descrierea problemelor ([specificații](#)), și
- să raționeze despre implementarea lor ([corecitudinea programelor](#)).

Care este comportamentul corect?

```
int main(void) {  
    int x = 0;  
    return (x = 1) + (x = 2);  
}
```

Care este comportamentul corect?

```
int main(void) {  
    int x = 0;  
    return (x = 1) + (x = 2);  
}
```

Conform standardului C, comportamentul programului este **nedefinit**.

- GCC4, MSVC: valoarea întoarsă e **4**
- GCC3, ICC, Clang: valoarea întoarsă e **3**

Care este comportamentul corect?

```
int r;
int f(int x) {
    return (r = x);
}
int main() {
    return f(1) + f(2), r;
}
```

Care este comportamentul corect?

```
int r;
int f(int x) {
    return (r = x);
}
int main() {
    return f(1) + f(2), r;
}
```

Conform standardului C, comportamentul programului este **corect**, dar **subspecificat**:
poate întoarce atât valoarea **1** cât și **2**.

Tipuri de semantică

Semantica dă "înțeles" unui program.

Tipuri de semantică

Semantica dă "înțeles" unui program.

- **Operațională:**

- Înțelesul programului este definit în funcție de pașii (transformări dintr-o stare în alta) care apar în timpul execuției.

Tipuri de semantică

Semantica dă "înțeles" unui program.

- **Operațională:**

- Înțelesul programului este definit în funcție de pașii (transformări dintr-o stare în alta) care apar în timpul execuției.

- **Denotațională:**

- Înțelesul programului este definit abstract ca element dintr-o structură matematică adecvată.

Tipuri de semantică

Semantica dă "înțeles" unui program.

- **Operațională:**

- Înțelesul programului este definit în funcție de pașii (transformări dintr-o stare în alta) care apar în timpul execuției.

- **Denotațională:**

- Înțelesul programului este definit abstract ca element dintr-o structură matematică adecvată.

- **Axiomatică:**

- Înțelesul programului este definit indirect în funcție de axiomele și regulile pe care le verifică.

Tipuri de semantică

Semantica dă "înțeles" unui program.

□ Operațională:

- Înțelesul programului este definit în funcție de pașii (transformări dintr-o stare în alta) care apar în timpul execuției.

□ Denotațională:

- Înțelesul programului este definit abstract ca element dintr-o structură matematică adecvată.

□ Axiomatică:

- Înțelesul programului este definit indirect în funcție de axiomele și regulile pe care le verifică.

□ Statică / a tipurilor

- Reguli de bună-formare pentru programe
- Oferă garanții privind execuția (e.g., nu se blochează)

Programare logică & Prolog

Programare logică - în mod idealist

- Un "program logic" este o colecție de proprietăți presupuse (sub formă de formule logice) despre lume (sau mai degrabă despre lumea programului).
- Programatorul furnizează și o proprietate (o formulă logică) care poate să fie sau nu adevărată în lumea respectivă ([întrebare](#), [query](#)).
- Sistemul determină dacă proprietatea aflată sub semnul întrebării este o consecință a proprietăților presupuse în program.
- Programatorul nu specifică metoda prin care sistemul verifică dacă întrebarea este sau nu consecință a programului.

Exemplu de program logic

```
oslo    →    windy
oslo    →    norway
norway  →    cold
cold ∧ windy  →    winterIsComing
                  oslo
```

Exemplu de program logic

```
oslo    →    windy
oslo    →    norway
norway  →    cold
cold ∧ windy  →    winterIsComing
                  oslo
```

Exemplu de întrebare

Este adevărat `winterIsComing`?

Prolog

- bazat pe logica clauzelor Horn
- semantica operațională este bazată pe rezoluție
- este Turing complet

Prolog

- bazat pe logica clauzelor Horn
- semantica operațională este bazată pe rezoluție
- este Turing complet

Limbajul Prolog este folosit pentru programarea sistemului IBM Watson!



Puteți citi mai multe detalii [aici](#).

Exemplul de mai sus în SWI-Prolog

Program:

```
windy :- oslo.  
norway :- oslo.  
cold :- norway.  
winterIsComing :- windy, cold.  
oslo.
```

Intrebare:

```
?- winterIsComing.  
true
```

<http://swish.swi-prolog.org/>

Quiz time!

<https://www.questionpro.com/t/AT4NiZrHFn>

Sintaxă: constante, variabile, termeni compuși

- **Atomi**: brian, 'Brian Griffin', brian_griffin

- **Numere**: 23, 23.03, -1

Atomii și numerele sunt constante.

- **Variabile**: X, Griffin, _family

- **Termeni compuși**: father(peter, stewie_griffin),
and(son(stewie,peter), daughter(meg,peter))

- forma generală: atom(termen,..., termen)
- atom-ul care denumește termenul se numește **functor**
- numărul de argumente se numește **aritate**



Un mic exercițiu sintactic

Care din următoarele siruri de caractere sunt **constante** și care sunt **variabile** în Prolog?

- vINCENT
- Footmassage
- variable23
- Variable2000
- big_kahuna_burger
- 'big kahuna burger'
- big kahuna burger
- 'Jules'
- _Jules
- '_Jules'

Un mic exercițiu sintactic

Care din următoarele siruri de caractere sunt **constante** și care sunt **variabile** în Prolog?

- vINCENT – **constantă**
- Footmassage – **variabilă**
- variable23 – **constantă**
- Variable2000 – **variabilă**
- big_kahuna_burger – **constantă**
- 'big kahuna burger' – **constantă**
- big kahuna burger – **nici una, nici alta**
- 'Jules' – **constantă**
- _Jules – **variabilă**
- '_Jules' – **constantă**

Program în Prolog = bază de cunoștințe

Example

Un program în Prolog:

```
father(peter,meg).  
father(peter,stewie).
```

```
mother(lois,meg).  
mother(lois,stewie).
```

```
griffin(peter).  
griffin(lois).
```

```
griffin(X) :- father(Y,X), griffin(Y).
```

Un program în Prolog este o **bază de cunoștințe** (Knowledge Base).

Program în Prolog = multime de predicate

Practic, gândim un program în Prolog ca o multime de **predicate** cu ajutorul cărora descriem *lumea (universul)* programului respectiv.

Example

```
father(peter,meg).  
father(peter,stewie).
```

```
mother(lois,meg).  
mother(lois,stewie).
```

```
griffin(peter).  
griffin(lois).
```

```
griffin(X) :- father(Y,X), griffin(Y).
```

Predicate:
father/2
mother/2
griffin/1

Un program în Prolog

Program

Fapte + Reguli

Program

- Un **program** în Prolog este format din **reguli** de forma
Head :- Body.
- **Head** este un predicat, iar **Body** este o secvență de predicate separate prin virgulă.
- Regulile fără Body se numesc **fapte**.

Program

- Un **program** în Prolog este format din **reguli** de forma
Head :- Body.
- **Head** este un predicat, iar **Body** este o secvență de predicate separate prin virgulă.
- Regulile fără Body se numesc **fapte**.

Example

- Exemplu de regulă: **griffin(X) :- father(Y,X), griffin(Y).**
- Exemplu de fapt: **father(peter,meg)** .

Interpretarea din punctul de vedere al logicii

- operatorul :- este implicația logică ←

Example

`comedy(X) :- griffin(X)`

dacă `griffin(X)` este adevărat, atunci `comedy(X)` este adevărat.

Interpretarea din punctul de vedere al logicii

- operatorul :- este implicația logică ←

Example

comedy(X) :- griffin(X)

dacă griffin(X) este adevărat, atunci comedy(X) este adevărat.

- virgula , este conjuncția ∧

Example

griffin(X) :- father(Y,X), griffin(Y).

dacă father(Y,X) și griffin(Y) sunt adevărate,
atunci griffin(X) este adevărat.

Interpretarea din punctul de vedere al logicii

- mai multe reguli cu **același Head** definesc același predicat, între definiții fiind un **sau** logic.

Example

```
comedy(X) :- family_guy(X).  
comedy(X) :- south_park(X).  
comedy(X) :- disenchantment(X).
```

dacă

family_guy(X) este adevărat sau south_park(X) este adevărat sau
disenchantment(X) este adevărat,

atunci

comedy(X) este adevărat.

Un program în Prolog

Program

Fapte + Reguli

Cum folosim un program în Prolog?

Întrebări în Prolog



Întrebări și ținte în Prolog

- Prolog poate răspunde la întrebări legate de consecințele relațiilor descrise într-un program în Prolog.
- Întrebările sunt de forma:
`?- predicat1(...),...,predicatn(...).`
- Prolog verifică dacă întrebarea este o consecință a relațiilor definite în program.
- Dacă este cazul, Prolog caută valori pentru variabilele care apar în întrebare astfel încât întrebarea să fie o consecință a relațiilor din program.
- Un predicat care este analizat pentru a se răspunde la o întrebare se numește **țintă (goal)**.

Întrebări în Prolog

Prolog poate da 2 tipuri de răspunsuri:

- **false** – în cazul în care întrebarea nu este o consecință a programului.
- **true** sau **valori pentru variabilele din întrebare** în cazul în care întrebarea este o consecință a programului.

Întrebări în Prolog

Prolog poate da 2 tipuri de răspunsuri:

- **false** – în cazul în care întrebarea nu este o consecință a programului.
- **true** sau **valori pentru variabilele din întrebare** în cazul în care întrebarea este o consecință a programului.

Example

```
?- griffin(meg)
true
?- griffin(glenn)
false
```

```
?- griffin(X)
X = petr ;
X = lois ;
X = meg ;
X = stewie ;
false
```

Pe săptămâna viitoare!

Curs 2

2021-2022

Fundamentele Limbajelor de Programare

Cuprins

1 Programare logică & Prolog

2 Liste și recursie

Programare logică & Prolog

Program în Prolog = multime de predicate

Practic, gândim un program în Prolog ca o multime de **predicate** cu ajutorul cărora descriem *lumea (universul)* programului respectiv.

Example

```
father(peter,meg).  
father(peter,stewie).
```

```
mother(lois,meg).  
mother(lois,stewie).
```

```
griffin(peter).  
griffin(lois).
```

```
griffin(X) :- father(Y,X), griffin(Y).
```

Predicate:

father/2
mother/2
griffin/1

Un program în Prolog

Program

Fapte + Reguli

Cum folosim un program în Prolog?

Întrebări în Prolog



Întrebări în Prolog

Prolog poate da 2 tipuri de răspunsuri:

- **false** – în cazul în care întrebarea nu este o consecință a programului.
- **true** sau **valori pentru variabilele din întrebare** în cazul în care întrebarea este o consecință a programului.

Example

```
?- griffin(meg)
true
?- griffin(glenn)
false
```

```
?- griffin(X)
X = petr ;
X = lois ;
X = meg ;
X = stewie ;
false
```

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, Prolog încearcă regulile în ordinea aparției lor.

Example

Să presupunem că avem programul:

```
foo(a).  foo(b).  foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).  
X = a.
```

Pentru a răspunde la întrebare se caută o potrivire (unificator) între scopul `foo(X)` și baza de cunoștințe. Răspunsul este substituția care realizează potrivirea, în cazul nostru `X = a`.

Răspunsul la întrebare este găsit prin unificare!

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, Prolog încearcă regulile în ordinea apariției lor.

Example

Să presupunem că avem programul:

```
foo(a).  foo(b).  foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).  
X = a.
```

```
?- foo(d).  
false
```

Dacă nu se poate face potrivirea, răspunsul este **false**.

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, Prolog încearcă regulile în ordinea apariției lor.

Example

Să presupunem că avem programul:

```
foo(a).  foo(b).  foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).  
X = a.
```

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, Prolog încearcă regulile în ordinea apariției lor.

Example

Să presupunem că avem programul:

```
foo(a).  foo(b).  foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).  
X = a.
```

Dacă dorim mai multe răspunsuri, tastăm ;

```
?- foo(X).  
X = a ;  
X = b ;  
X = c.
```

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, Prolog încearcă regulile în ordinea apariției lor.

Example

Să presupunem că avem programul:

```
foo(a).  
foo(b).  
foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).
```

```
?- trace.  
true.  
  
[trace] ?- foo(X).  
Call: (8) foo(_4556) ? creep  
Exit: (8) foo(a) ? creep  
X = a ;  
Redo: (8) foo(_4556) ? creep  
Exit: (8) foo(b) ? creep  
X = b ;  
Redo: (8) foo(_4556) ? creep  
Exit: (8) foo(c) ? creep  
X = c.
```

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, Prolog redenumește variabilele.

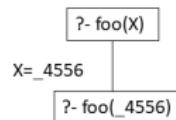
Example

Să presupunem că avem programul:

```
foo(a).  
foo(b).  
foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).
```



Cum găsește Prolog răspunsul

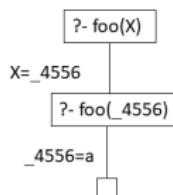
Pentru a găsi un răspuns, Prolog încearcă regulile în ordinea apariției lor.

Example

Să presupunem că avem programul:

```
foo(a).  
foo(b).  
foo(c).
```

și că punem următoarea întrebare:
`?- foo(X).`



În acest moment, a fost găsită prima soluție: `X=_4556=a`.

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns, Prolog încearcă clauzele în ordinea apariției lor.

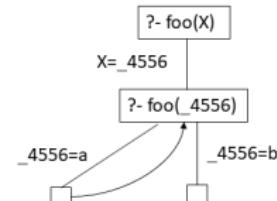
Example

Să presupunem că avem programul:

```
foo(a).  
foo(b).  
foo(c).
```

și că punem următoarea întrebare:

```
?- foo(X).
```



Dacă se dorește încă un răspuns, atunci se face un pas înapoi în [arborele de căutare](#) și se încearcă satisfacerea ţintei cu o nouă valoare.

Cum găsește Prolog răspunsul

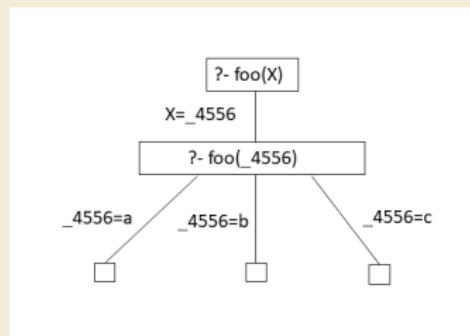
Pentru a găsi un răspuns, Prolog încearcă clauzele în ordinea apariției lor.

Example

Să presupunem că avem programul:

```
foo(a).  
foo(b).  
foo(c).
```

și că punem următoarea întrebare:
?- foo(X).



arborele de căutare

Cum găsește Prolog răspunsul

Example

Să presupunem că avem programul:

```
bar(b).  
bar(c).  
baz(c).
```

și că punem următoarea întrebare:
?- bar(X), baz(X).



Cum găsește Prolog răspunsul

Prolog se întoarce la ultima alegere dacă o sub-țintă eșuează.

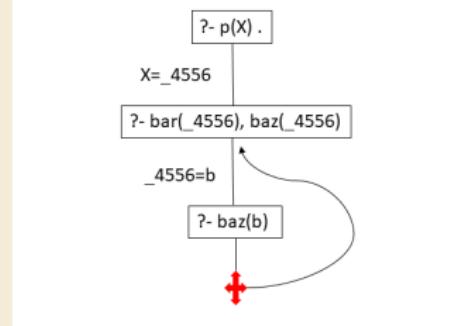
Example

Să presupunem că avem programul:

```
bar(b).  
bar(c).  
baz(c).
```

și că punem următoarea întrebare:

```
?- bar(X), baz(X).
```



Cum găsește Prolog răspunsul

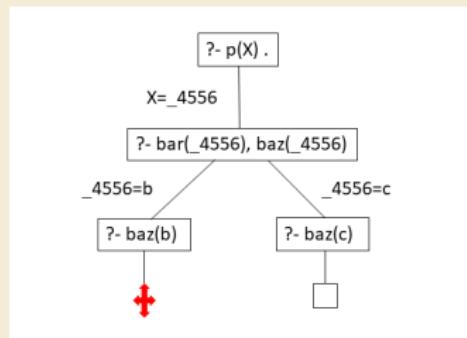
Prolog se întoarce la ultima alegere dacă o sub-tintă eşuează.

Example

Să presupunem că avem programul:

```
bar(b).  
bar(c).  
baz(c).
```

și că punem următoarea întrebare:
?- bar(X),baz(X).



Soluția găsită este: $X=_4556=c$.

Cum găsește Prolog răspunsul

Ce se întâmplă dacă schimbăm ordinea regulilor?

Example

Să presupunem că avem programul:

```
bar(c).  
bar(b).  
baz(c).
```

și că punem următoarea întrebare:

```
?- bar(X), baz(X).
```

Cum găsește Prolog răspunsul

Ce se întâmplă dacă schimbăm ordinea regulilor?

Example

Să presupunem că avem programul:

```
bar(c).
```

```
bar(b).
```

```
baz(c).
```

și că punem următoarea întrebare:

```
?- bar(X), baz(X).
```

```
X = c ;
```

```
false
```

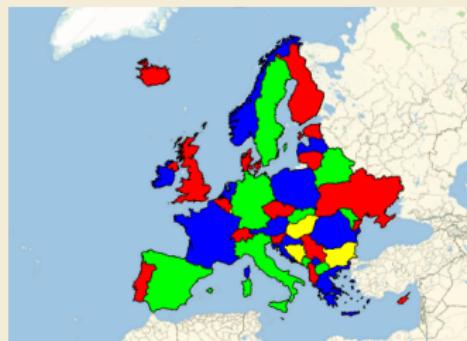
Vă explicați ce s-a întâmplat? Desenați arborele de căutare!

Un program mai complicat

Problema colorării hărților

Să se coloreze o hartă dată cu un număr minim de culori astfel încât oricare două țări vecine să fie colorate diferit.

Example



Sursa imaginii

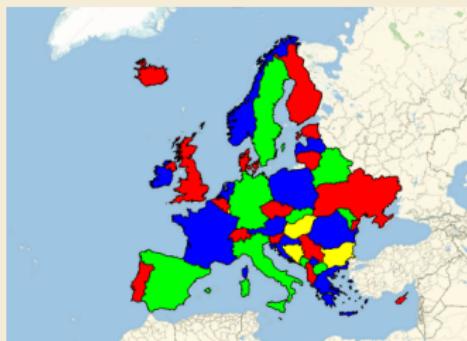
Un program mai complicat

Problema colorării hărților

Să se coloreze o hartă dată cu un număr minim de culori astfel încât oricare două țări vecine să fie colorate diferit.

Cum modelăm această problemă în Prolog?

Example



Sursa imaginii

Un program mai complicat

Problema colorării hărților

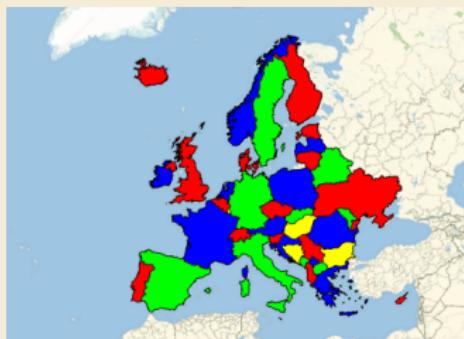
Să se coloreze o hartă dată cu un număr minim de culori astfel încât oricare două țări vecine să fie colorate diferit.

Cum modelăm această problemă în Prolog?

Example

Trebuie să definim:

- culorile
- harta
- constrângerile



Sursa imaginii

Problema colorării hărților

Definim culorile

Example

```
culoare(albastru).  
culoare(rosu).  
culoare(verde).  
culoare(galben).
```

Problema colorării hărților

Definim culorile, harta

Example

```
culoare(albastru).  
culoare(rosu).  
culoare(verde).  
culoare(galben).
```

```
harta(R0,SE,MD,UA,BG,HU) :- vecin(R0,SE), vecin(R0,UA),  
                                vecin(R0,MD), vecin(R0,BG),  
                                vecin(R0,HU), vecin(UA,MD),  
                                vecin(BG,SE), vecin(SE,HU).
```

Problema colorării hărților

Definim culorile, harta și constrângerile.

Example

```
culoare(albastru).
```

```
culoare(rosu).
```

```
culoare(verde).
```

```
culoare(galben).
```

```
harta(R0, SE, MD, UA, BG, HU) :- vecin(R0, SE), vecin(R0, UA),
                                         vecin(R0, MD), vecin(R0, BG),
                                         vecin(R0, HU), vecin(UA, MD),
                                         vecin(BG, SE), vecin(SE, HU).
```

```
vecin(X, Y) :- culoare(X),
                  culoare(Y),
                  X \== Y.
```

Problema colorării hărților

Definim culorile, harta și constrângerile. Cum punem întrebarea?

Example

```
culoare(albastru).  
culoare(rosu).  
culoare(verde).  
culoare(galben).
```

```
harta(R0, SE, MD, UA, BG, HU) :- vecin(R0, SE), vecin(R0, UA),  
                                     vecin(R0, MD), vecin(R0, BG),  
                                     vecin(R0, HU), vecin(UA, MD),  
                                     vecin(BG, SE), vecin(SE, HU).
```

```
vecin(X, Y) :- culoare(X),  
              culoare(Y),  
              X \== Y.
```

Problema colorării hărților

Definim culorile, harta și constrângerile. Cum punem întrebarea?

Example

```
culoare(albastru).
```

```
culoare(rosu).
```

```
culoare(verde).
```

```
culoare(galben).
```

```
harta(R0, SE, MD, UA, BG, HU) :- vecin(R0, SE), vecin(R0, UA),
                                         vecin(R0, MD), vecin(R0, BG),
                                         vecin(R0, HU), vecin(UA, MD),
                                         vecin(BG, SE), vecin(SE, HU).
```

```
vecin(X, Y) :- culoare(X),
                  culoare(Y),
                  X \== Y.
```

```
?- harta(R0, SE, MD, UA, BG, HU).
```

Problema colorării hărților

Ce răspuns primim?

Example

```
culoare(albastru).  
culoare(rosu).  
culoare(verde).  
culoare(galben).
```

```
harta(R0, SE, MD, UA, BG, HU) :- vecin(R0, SE), vecin(R0, UA),  
                                     vecin(R0, MD), vecin(R0, BG),  
                                     vecin(R0, HU), vecin(UA, MD),  
                                     vecin(BG, SE), vecin(SE, HU).
```

```
vecin(X, Y) :- culoare(X),  
              culoare(Y),  
              X \== Y.
```

```
?- harta(R0, SE, MD, UA, BG, HU).
```

Problema colorării hărților

Example

```
culoare(albastru).
```

```
culoare(rosu).
```

```
culoare(verde).
```

```
culoare(galben).
```

```
harta(R0, SE, MD, UA, BG, HU) :- vecin(R0, SE), vecin(R0, UA),
                                         vecin(R0, MD), vecin(R0, BG),
                                         vecin(R0, HU), vecin(UA, MD),
                                         vecin(BG, SE), vecin(SE, HU).
```

```
vecin(X, Y) :- culoare(X),
               culoare(Y),
               X \== Y.
```

```
?- harta(R0, SE, MD, UA, BG, HU).
```

```
R0 = albastru,
```

```
SE = UA, UA = rosu,
```

```
MD = BG, BG = HU, HU = verde ■
```

Compararea termenilor: $=$, $\backslash=$, $==$, $\backslash==$

- $T = U$ reușește dacă există o potrivire (termenii se unifică)
- $T \backslash= U$ reușește dacă nu există o potrivire
- $T == U$ reușește dacă termenii sunt identici
- $T \backslash== U$ reușește dacă termenii sunt diferiți

Compararea termenilor: `=`, `\=`, `==`, `\==`

<code>T = U</code>	reușește dacă există o potrivire (termenii se unifică)
<code>T \= U</code>	reușește dacă nu există o potrivire
<code>T == U</code>	reușește dacă termenii sunt identici
<code>T \== U</code>	reușește dacă termenii sunt diferiți

Example

?- `X = Y.`

`X = Y.`

?- `X == Y .`

`false`

?- `p(X,q(Z)) = p(Y,X) .`

`X = Y, Y = q(Z) .`

?- `p(X,Y) == p(X,Y) .`

`true`

?- `2 = 1 + 1`

`false`

?- `2 == 1 + 1`

`false`

- În exemplul de mai sus, `1+1` este privită ca o expresie, nu este evaluată. Există și predicate care forțează evaluarea (e.g., `=:=`).

Negarea unui predicat: $\setminus+$ pred(X)

Example

```
animal(dog). animal(elephant). animal(sheep).
```

```
?- animal(cat).
```

```
false
```

```
?-  $\setminus+$  animal(cat).
```

```
true
```

Negarea unui predicat: \+ pred(X)

Example

```
animal(dog). animal(elephant). animal(sheep).
```

```
?- animal(cat).
```

```
false
```

```
?- \+ animal(cat).
```

```
true
```

- Clauzele din Prolog dau doar condiții suficiente, dar nu și necesare pentru ca un predicat să fie adevărat.
- Pentru a da un răspuns pozitiv la o întă, Prolog trebuie să construiască o "demonstrație" pentru a arată că mulțimea de fapte și reguli din program implică acea întă.
- Astfel, un răspuns **false** nu înseamnă neapărat că întă nu este adevărată, ci doar că **Prolog nu a reușit să găsească o demonstrație**.

Operatorul \+

- Negarea unei ţinte se poate defini astfel:

```
neg(Goal) :- Goal, !, fail.  
neg(Goal)
```

unde **fail/0** este un predicat care eşuează întotdeauna.

Operatorul \+

- Negarea unei ţinte se poate defini astfel:

```
neg(Goal) :- Goal, !, fail.  
neg(Goal)
```

unde `fail/0` este un predicat care eşuează întotdeauna.

- În PROLOG acest predicat este predefinit sub numele \+.
- Operatorul \+ se folosește pentru a nega un predicat.
- `!(cut)` este un predicat predefinit (de aritate 0) care restricționează mecanismul de backtracking: execuția subținței `!` se termină cu succes, deci alegerile (instantierile) făcute înainte de a se ajunge la `!` nu mai pot fi schimbată.
- O ţintă \+ Goal reușește dacă Prolog nu găsește o demonstrație pentru Goal. Negarea din Prolog este definită ca incapacitatea de a găsi o demonstrație.
- Semantica operatorului \+ se numește **negation as failure**.

Negăția ca eșec ("negation as failure")

Example

Să presupunem că avem o listă de fapte cu perechi de oameni căsătoriți
între ei:

```
married(peter, lucy).  
married(paul, mary).  
married(bob, juliet).  
married(harry, geraldine).
```

Negăția ca eșec

Example (cont.)

Putem să definim un predicat `single/1` care reușește dacă argumentul său nu este nici primul nici al doilea argument în faptele pentru `married`.

```
single(Person) :-
```

```
  \+ married(Person, _),  
  \+ married(_, Person).
```

```
?- single(mary).    ?- single(anne).    ?- single(X).  
false           true            false
```

Răspunsul la întrebarea `?- single(anne).` trebuie gândit astfel:

Presupunem că Anne este single,
deoarece nu am putut demonstra că este maritată.

Predicatul -> /2 (if-then-else)

□ if-then

```
If -> Then :- If, !, Then.
```

Predicatul -> /2 (if-then-else)

- if-then

```
If -> Then :- If, !, Then.
```

- if-then-else

```
If -> Then; _Else :- If, !, Then.
```

```
If -> Then; Else :- !, Else.
```

Se încearcă demonstrarea predicatului If. Dacă întoarce true atunci se încearcă demonstrarea predicatului Then, iar dacă întoarce false se încearcă demonstrarea predicatului Else.

```
max(X,Y,Z) :- (X <= Y) -> Z = Y ; Z = X
```

```
?- max(2,3,Z).
```

```
Z = 3.
```

Predicatul $\rightarrow /2$ (if-then-else)

- if-then

```
If -> Then :- If, !, Then.
```

- if-then-else

```
If -> Then; _Else :- If, !, Then.
```

```
If -> Then; Else :- !, Else.
```

Se încearcă demonstrarea predicatului If. Dacă întoarce true atunci se încearcă demonstrarea predicatului Then, iar dacă întoarce false se încearcă demonstrarea predicatului Else.

```
max(X,Y,Z) :- (X <= Y) -> Z = Y ; Z = X
```

```
?- max(2,3,Z).
```

```
Z = 3.
```

Observăm că If \rightarrow Then este echivalent cu If \rightarrow Then ; fail.

Liste și recursie

Listă [t₁, ..., t_n]

- O listă în Prolog este un sir de elemente, separate prin virgulă, între paranteze drepte:

[1, cold, parent(jon), [winter, is, coming], X]

- O listă poate contine termeni de orice fel.
- Ordinea termenilor din listă are importanță:

?- [1,2] == [2,1] .

false

- Lista vidă se notează [].
- Simbolul | desemnează coada listei:

?- [1,2,3,4,5,6] = [X|T].

X = 1, T = [2, 3, 4, 5, 6].

?- [1,2,3|[4,5,6]] == [1,2,3,4,5,6].

true.

Listă [t₁, ..., t_n] == [t₁ | [t₂, ..., t_n]]

- Simbolul | desemnează coada listei:

```
?- [1,2,3,4,5,6] = [X|T].  
X = 1,  
T = [2, 3, 4, 5, 6].
```

- Variabila anonimă _ este unificată cu orice termen Prolog:

```
?- [1,2,3,4,5,6] = [X|_].  
X = 1.
```

- Deoarece Prologul face unificare poate identifica şabloane mai complicate:

```
?- [5,1,1,3,2]=[_|[X|[X|_]]].  
X = 1.  
?- [5,1,4,3,2]=[_|[X|[X|_]]].  
false.
```

Liste

Exercițiu

- Definiți un predicat care verifică că un termen este lista.

Exercițiu

- Definiți un predicat care verifică că un termen este lista.

```
is_list([]).  
is_list([_|_]).
```

Liste

Exercițiu

- Definiți un predicat care verifică că un termen este lista.

```
is_list([]).  
is_list([_|_]).
```

- Definiți predicate care verifică dacă un termen este primul element, ultimul element sau coada unei liste.

Liste

Exercițiu

- Definiți un predicat care verifică că un termen este lista.

```
is_list([]).  
is_list([_|_]).
```

- Definiți predicate care verifică dacă un termen este primul element, ultimul element sau coada unei liste.

```
head([X|_], X).
```

```
last([X], X).  
last([_|T], Y) :- last(T, Y).
```

```
tail([], []).  
tail([_|T], T).
```

Exercițiu

- Definiți un predicat care verifică dacă un termen aparține unei liste.

Exercițiu

- Definiți un predicat care verifică dacă un termen aparține unei liste.

```
member(H, [H|_]).
```

```
member(H, [_|T]) :- member(H, T).
```

Liste

Exercițiu

- Definiți un predicat care verifică dacă un termen aparține unei liste.

```
member(H, [H|_]).
```

```
member(H, [_|T]) :- member(H, T).
```

- Definiți un predicat append/3 care verifică dacă o listă se obține prin concatenarea altor două liste.

Liste

Exercițiu

- Definiți un predicat care verifică dacă un termen aparține unei liste.

```
member(H, [H|_]).
```

```
member(H, [_|T]) :- member(H,T).
```

- Definiți un predicat append/3 care verifică dacă o listă se obține prin concatenarea altor două liste.

```
append([], L, L).
```

```
append([X|T], L, [X|R]) :- append(T, L, R).
```

Liste

Exercițiu

- Definiți un predicat care verifică dacă un termen aparține unei liste.

```
member(H, [H|_]).
```

```
member(H, [_|T]) :- member(H, T).
```

- Definiți un predicat append/3 care verifică dacă o listă se obține prin concatenarea altor două liste.

```
append([], L, L).
```

```
append([X|T], L, [X|R]) :- append(T, L, R).
```

Există predicatele predefinite `member/2` și `append/3`.

Liste append/3

- Funcția append/3:

```
?- listing	append/3).
```

```
append([],L,L).
```

```
append([X|T],L, [X|R]) :- append(T,L,R).
```

```
?- append(X,Y,[a,b,c]).
```

```
X = [],
```

```
Y = [a, b, c] ;
```

```
X = [a],
```

```
Y = [b, c] ;
```

```
X = [a, b],
```

```
Y = [c] ;
```

```
X = [a, b, c],
```

```
Y = [] ;
```

false

- Funcția astfel definită poate fi folosită atât pentru verificare, cât și pentru generare.

Exercițiu

- Definiți un predicat `elim/3` care verifică dacă o listă se obține din alta prin eliminarea unui element.

Exercițiu

- Definiți un predicat `elim/3` care verifică dacă o listă se obține din alta prin eliminarea unui element.

```
elim(X, [X|T], T).
```

```
elim(X, [H|T], [H|L]) :- elim(X, T, L).
```

Liste

Exercițiu

- Definiți un predicat `elim/3` care verifică dacă o listă se obține din alta prin eliminarea unui element.

```
elim(X, [X|T], T).
```

```
elim(X, [H|T], [H|L]) :- elim(X, T, L).
```

- Definiți un predicat care `perm/2` care verifică dacă două liste sunt permutări.

Liste

Exercițiu

- Definiți un predicat `elim/3` care verifică dacă o listă se obține din alta prin eliminarea unui element.

```
elim(X, [X|T], T).
```

```
elim(X, [H|T], [H|L]) :- elim(X, T, L).
```

- Definiți un predicat care `perm/2` care verifică dacă două liste sunt permutări.

```
perm([], []).
```

```
perm([X|T], L) :- elim(X, L, R), perm(R, T).
```

Liste

Exercițiu

- Definiți un predicat `elim/3` care verifică dacă o listă se obține din alta prin eliminarea unui element.

```
elim(X, [X|T], T).
```

```
elim(X, [H|T], [H|L]) :- elim(X, T, L).
```

- Definiți un predicat care `perm/2` care verifică dacă două liste sunt permutări.

```
perm([], []).
```

```
perm([X|T], L) :- elim(X, L, R), perm(R, T).
```

Predicatelor predefinite `select/3` și `permutation/2` au aceeași funcționalitate.

Quiz time!

<https://www.questionpro.com/t/AT4NiZrPCD>



Pe săptămâna viitoare!

Curs 3

2021-2022

Programare Logică

Cuprins

1 Substituții și unificare

Substituții și unificare

Termeni

Alfabet:

- \mathcal{F} o multime de simboluri de functii de aritate cunoscuta
- \mathcal{V} o multime numarabila de variabile
- \mathcal{F} si \mathcal{V} sunt disjuncte

Termeni

Alfabet:

- \mathcal{F} o multime de simboluri de functii de aritate cunoscuta
- \mathcal{V} o multime numarabila de variabile
- \mathcal{F} si \mathcal{V} sunt disjuncte

Termeni peste \mathcal{F} si \mathcal{V} :

$$t ::= x \mid f(t_1, \dots, t_n)$$

unde

- $n \geq 0$
- x este o variabila
- f este un simbol de functie de aritate n

Termeni

Notatii:

- constante**: simboluri de functii de aritate 0
- x, y, z, \dots pentru variabile
- a, b, c, \dots pentru constante
- f, g, h, \dots pentru simboluri de functii arbitrate
- s, t, u, \dots pentru termeni
- $\text{var}(t)$ multimea variabilelor care apa in t
- ecuatii $s = t$ pentru o pereche de termeni
- $\text{Trm}_{\mathcal{F}, \mathcal{V}}$ multimea termenilor peste \mathcal{F} si \mathcal{V}

Termeni

Exemplu

- $f(x, g(x, a), y)$ este un termen, unde f are aritate 3, g are aritate 2, a este o constantă
- $\text{var}(f(x, g(x, a), y)) = \{x, y\}$

Substituții

Definiție

O **substituție** σ este o funcție (parțială) de la variabile la termeni, adică

$$\sigma : \mathcal{V} \rightarrow \text{Trm}_{\mathcal{F}, \mathcal{V}}$$

Exemplu

În notația uzuală, $\sigma = \{x/a, y/g(w), z/b\}$. Substitutia σ este identitate pe restul variabilelor.

Notatie alternativa $\sigma = \{x \mapsto a, y \mapsto g(w), z \mapsto b\}$.

Substituții

- Substituțiile sunt o modalitate de a înlocui variabilele cu alți termeni.
- Substituțiile se aplică simultan pe toate variabilele.

Aplicarea unei substituții σ unui termen t :

$$\sigma(t) = \begin{cases} \sigma(x), & \text{daca } t = x \\ f(\sigma(t_1), \dots, \sigma(t_n)), & \text{daca } t = f(t_1, \dots, t_n) \end{cases} .$$

Substituții

- Substituțiile sunt o modalitate de a înlocui variabilele cu alți termeni.
- Substituțiile se aplică simultan pe toate variabilele.

Aplicarea unei substituții σ unui termen t :

$$\sigma(t) = \begin{cases} \sigma(x), & \text{daca } t = x \\ f(\sigma(t_1), \dots, \sigma(t_n)), & \text{daca } t = f(t_1, \dots, t_n) \end{cases} .$$

Exemplu

- $\sigma = \{x \mapsto f(x, y), y \mapsto g(a)\}$
- $t = f(x, g(f(f(x, f(y, z)))))$
- $\sigma(t) = f(f(x, y), g(f(f(x, y), f(g(a), z))))$

Substituții

Două substituții σ_1 și σ_2 se pot compune

$\sigma_1; \sigma_2$

(aplicăm întâi σ_1 , apoi σ_2).

Substituții

Două substituții σ_1 și σ_2 se pot compune

$\sigma_1; \sigma_2$

(aplicăm întâi σ_1 , apoi σ_2).

Exemplu

□ $t = h(u, v, x, y, z)$

Substituții

Două substituții σ_1 și σ_2 se pot compune

$\sigma_1; \sigma_2$

(aplicăm întâi σ_1 , apoi σ_2).

Exemplu

- $t = h(u, v, x, y, z)$
- $\tau = \{x \mapsto f(y), y \mapsto f(a), z \mapsto u\}$
- $\sigma = \{y \mapsto g(a), u \mapsto z, v \mapsto f(f(a))\}$

Substituții

Două substituții σ_1 și σ_2 se pot compune

$\sigma_1; \sigma_2$

(aplicăm întâi σ_1 , apoi σ_2).

Exemplu

- $t = h(u, v, x, y, z)$
- $\tau = \{x \mapsto f(y), y \mapsto f(a), z \mapsto u\}$
- $\sigma = \{y \mapsto g(a), u \mapsto z, v \mapsto f(f(a))\}$
- $(\tau; \sigma)(t) = \sigma(\tau(t)) = \sigma(h(u, v, f(y), f(a), u)) =$
 $= h(z, f(f(a)), f(g(a)), f(a), z)$

Substituții

Două substituții σ_1 și σ_2 se pot compune

$\sigma_1; \sigma_2$

(aplicăm întâi σ_1 , apoi σ_2).

Exemplu

- $t = h(u, v, x, y, z)$
- $\tau = \{x \mapsto f(y), y \mapsto f(a), z \mapsto u\}$
- $\sigma = \{y \mapsto g(a), u \mapsto z, v \mapsto f(f(a))\}$
- $(\tau; \sigma)(t) = \sigma(\tau(t)) = \sigma(h(u, v, f(y), f(a), u)) =$
 $= h(z, f(f(a)), f(g(a)), f(a), z)$
- $(\sigma; \tau)(t) = \tau(\sigma(t)) = \tau(h(z, f(f(a)), x, g(a), z))$
 $= h(u, f(f(a)), f(y), g(a), u)$

Unificare

- Doi termeni t_1 și t_2 **se unifică** dacă există o substituție σ astfel încât $\sigma(t_1) = \sigma(t_2)$.
- În acest caz, σ se numește **unificatorul** termenilor t_1 și t_2 .

Unificare

- Doi termeni t_1 și t_2 **se unifică** dacă există o substituție σ astfel încât $\sigma(t_1) = \sigma(t_2)$.
- În acest caz, σ se numește **unificatorul** termenilor t_1 și t_2 .
- În programarea logică, unificatorii sunt ingredientele de bază în execuția unui program.

Unificare

- Doi termeni t_1 și t_2 se unifică dacă există o substituție σ astfel încât
$$\sigma(t_1) = \sigma(t_2).$$
- În acest caz, σ se numește unificatorul termenilor t_1 și t_2 .
- În programarea logică, unificatorii sunt ingredientele de bază în execuția unui program.
- Un unificator σ pentru t_1 și t_2 este un cel mai general unificator (cgu,mgu) dacă pentru orice alt unificator σ' pentru t_1 și t_2 , există o substituție τ astfel încât

$$\sigma' = \sigma; \tau.$$

Unificator

Exemplu

- $t = x + (y * y) = +(x, *(y, y))$
- $t' = x + (y * x) = +(x, *(y, x))$

Unificator

Exemplu

- $t = x + (y * y) = +(x, *(y, y))$
- $t' = x + (y * x) = +(x, *(y, x))$
- $\sigma = \{x/y, y/y\}$
 - $\sigma(t) = y + (y * y)$
 - $\sigma(t') = y + (y * y)$
 - σ este cgu

Unificator

Exemplu

- $t = x + (y * y) = +(x, *(y, y))$
- $t' = x + (y * x) = +(x, *(y, x))$
- $\sigma = \{x/y, y/y\}$
 - $\sigma(t) = y + (y * y)$
 - $\sigma(t') = y + (y * y)$
 - σ este cgu
- $\sigma' = \{x/0, y/0\}$
 - $\sigma'(t) = 0 + (0 * 0)$
 - $\sigma'(t') = 0 + (0 * 0)$

Unificator

Exemplu

- $t = x + (y * y) = +(x, *(y, y))$
- $t' = x + (y * x) = +(x, *(y, x))$
- $\sigma = \{x/y, y/y\}$
 - $\sigma(t) = y + (y * y)$
 - $\sigma(t') = y + (y * y)$
 - σ este cgu
- $\sigma' = \{x/0, y/0\}$
 - $\sigma'(t) = 0 + (0 * 0)$
 - $\sigma'(t') = 0 + (0 * 0)$
 - $\sigma' = \sigma; \{y/0\}$

Unificator

Exemplu

- $t = x + (y * y) = +(x, *(y, y))$
- $t' = x + (y * x) = +(x, *(y, x))$
- $\sigma = \{x/y, y/y\}$
 - $\sigma(t) = y + (y * y)$
 - $\sigma(t') = y + (y * y)$
 - σ este cgu
- $\sigma' = \{x/0, y/0\}$
 - $\sigma'(t) = 0 + (0 * 0)$
 - $\sigma'(t') = 0 + (0 * 0)$
 - $\sigma' = \sigma; \{y/0\}$
 - σ' este unificator, dar nu este cgu

Algoritmul de unificare

- Pentru o mulțime finită de termeni $\{t_1, \dots, t_n\}$, $n \geq 2$, algoritmul de unificare stabilește dacă există un cgu.
- Există algoritmi mai eficienți, dar îl alegem pe acesta pentru simplitatea sa.

Algoritmul de unificare

- Pentru o mulțime finită de termeni $\{t_1, \dots, t_n\}$, $n \geq 2$, algoritmul de unificare stabilește dacă există un cgu.
- Există algoritmi mai eficienți, dar îl alegem pe acesta pentru simplitatea sa.
- Algoritmul lucrează cu două liste:
 - Lista soluție: S
 - Lista de rezolvat: R

Algoritmul de unificare

- Pentru o mulțime finită de termeni $\{t_1, \dots, t_n\}$, $n \geq 2$, algoritmul de **unificare** stabilește dacă există un cgu.
- Există algoritmi mai eficienți, dar îl alegem pe acesta pentru simplitatea sa.
- Algoritmul lucrează cu două liste:
 - Lista soluție: S
 - Lista de rezolvat: R
- Inițial:
 - Lista soluție: $S = \emptyset$
 - Lista de rezolvat: $R = \{t_1 \doteq t_2, \dots, t_{n-1} \doteq t_n\}$

Algoritmul de unificare

Algoritmul constă în aplicarea regulilor de mai jos:

Algoritmul de unificare

Algoritmul constă în aplicarea regulilor de mai jos:

- **SCOATE**

- orice ecuație de forma $t \doteq t$ din \mathcal{R} este **eliminată**.

Algoritmul de unificare

Algoritmul constă în aplicarea regulilor de mai jos:

- **SCOATE**

- orice ecuație de forma $t \doteq t$ din \mathcal{R} este **eliminată**.

- **DESCOMPUNE**

- orice ecuație de forma $f(t_1, \dots, t_n) \doteq f(t'_1, \dots, t'_n)$ din \mathcal{R} este **înlocuită** cu ecuațiile $t_1 \doteq t'_1, \dots, t_n \doteq t'_n$.

Algoritmul de unificare

Algoritmul constă în aplicarea regulilor de mai jos:

□ SCOATE

- orice ecuație de forma $t \doteq t$ din R este **eliminată**.

□ DESCOMPUNE

- orice ecuație de forma $f(t_1, \dots, t_n) \doteq f(t'_1, \dots, t'_n)$ din R este **înlocuită** cu ecuațiile $t_1 \doteq t'_1, \dots, t_n \doteq t'_n$.

□ REZOLVĂ

- orice ecuație de forma $x \doteq t$ sau $t \doteq x$ din R , unde variabila x nu apare **în termenul** t , este **mutată** sub forma $x \doteq t$ în S .
În toate celelalte ecuații (din R și S), x este **înlocuit cu** t .

Algoritmul de unificare

Algoritmul se termină normal dacă $R = \emptyset$. În acest caz, S conține cgu.

Algoritmul de unificare

Algoritmul se termină normal dacă $R = \emptyset$. În acest caz, S conține cgu.

Algoritmul este oprit cu concluzia inexistenței unui cgu dacă:

- 1 În R există o ecuație de forma

$$f(t_1, \dots, t_n) \doteq g(t'_1, \dots, t'_k) \text{ cu } f \neq g.$$

- 2 În R există o ecuație de forma $x \doteq t$ sau $t \doteq x$ și variabila x apare în termenul t .

Algoritmul de unificare - schemă

	Lista soluție S	Lista de rezolvat R
Inițial	\emptyset	$t_1 \doteq t'_1, \dots, t_n \doteq t'_n$
SCOATE	S	$R', t \doteq t$
	S	R'
DESCOMPUNE	S	$R', f(t_1, \dots, t_n) \doteq f(t'_1, \dots, t'_n)$
	S	$R', t_1 \doteq t'_1, \dots, t_n \doteq t'_n$
REZOLVĂ	S	$R', x \doteq t$ sau $t \doteq x$, x nu apare în t
	$x \doteq t, S[x/t]$	$R'[x/t]$
Final	S	\emptyset

$S[x/t]$: în toate ecuațiile din S , x este înlocuit cu t

Exemplu

Exemplu

- Ecuatiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)\}$ au cgu?

Exemplu

Exemplu

- Ecuatiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)\}$ au cgu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)$	

Exemplu

Exemplu

- Ecuatiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)\}$ au cgu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)$	REZOLVĂ

Exemplu

Exemplu

□ Ecuatiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)\}$ au cgu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(g(y)), y) \doteq f(g(z), w, z)$	

Exemplu

Exemplu

- Ecuățiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)\}$ au cgu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(g(y)), y) \doteq f(g(z), w, z)$	DESCOMPUNE

Exemplu

Exemplu

- Ecuățiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)\}$ au cgu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(g(y)), y) \doteq f(g(z), w, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq g(z), h(g(y)) \doteq w, y \doteq z$	

Exemplu

Exemplu

- Ecuățiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)\}$ au cgu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(g(y)), y) \doteq f(g(z), w, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq g(z), h(g(y)) \doteq w, y \doteq z$	REZOLVĂ

Exemplu

Exemplu

- Ecuățiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)\}$ au cgu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(g(y)), y) \doteq f(g(z), w, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq g(z), h(g(y)) \doteq w, y \doteq z$	REZOLVĂ
$w \doteq h(g(y)),$ $x \doteq g(y)$	$g(y) \doteq g(z), y \doteq z$	REZOLVĂ

Exemplu

Exemplu

- Ecuatiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)\}$ au cgu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(g(y)), y) \doteq f(g(z), w, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq g(z), h(g(y)) \doteq w, y \doteq z$	REZOLVĂ
$w \doteq h(g(y)),$ $x \doteq g(y)$	$g(y) \doteq g(z), y \doteq z$	REZOLVĂ
$y \doteq z, x \doteq g(z),$ $w \doteq h(g(z))$	$g(z) \doteq g(z)$	SCOATE

Exemplu

Exemplu

- Ecuatiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)\}$ au cgu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(x), y) \doteq f(g(z), w, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(g(y)), y) \doteq f(g(z), w, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq g(z), h(g(y)) \doteq w, y \doteq z$	REZOLVĂ
$w \doteq h(g(y)),$ $x \doteq g(y)$	$g(y) \doteq g(z), y \doteq z$	REZOLVĂ
$y \doteq z, x \doteq g(z),$ $w \doteq h(g(z))$	$g(z) \doteq g(z)$	SCOATE
$y \doteq z, x \doteq g(z),$ $w \doteq h(g(z))$	\emptyset	

- $\sigma = \{y \mapsto z, x \mapsto g(z), w \mapsto h(g(z))\}$ este cgu.

Exemplu

Exemplu

- Ecuațiile $\{g(y) \dot{=} x, f(x, h(y), y) \dot{=} f(g(z), b, z)\}$ au cgu?

Exemplu

Exemplu

□ Ecuațiile $\{g(y) \doteq x, f(x, h(y), y) \doteq f(g(z), b, z)\}$ au cgu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(y), y) \doteq f(g(z), b, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(y), y) \doteq f(g(z), b, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq g(z), h(y) \doteq b, y \doteq z$	- EŞEC -

Exemplu

Exemplu

- Ecuațiile $\{g(y) \doteq x, f(x, h(y), y) \doteq f(g(z), b, z)\}$ au cgu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(y), y) \doteq f(g(z), b, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(y), y) \doteq f(g(z), b, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq g(z), h(y) \doteq b, y \doteq z$	- EŞEC -

- h și b sunt simboluri de operații diferite!
- Nu există unificator pentru acești termeni.

Exemplu

Exemplu

- Ecuațiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(y, w, z)\}$ au cgu?

Exemplu

Exemplu

□ Ecuațiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(y, w, z)\}$ au cgu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(x), y) \doteq f(y, w, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(g(y)), y) \doteq f(y, w, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq y, h(g(y)) \doteq w, y \doteq z$	- EŞEC -

Exemplu

Exemplu

- Ecuațiile $\{g(y) \doteq x, f(x, h(x), y) \doteq f(y, w, z)\}$ au cgu?

S	R	
\emptyset	$g(y) \doteq x, f(x, h(x), y) \doteq f(y, w, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(g(y)), y) \doteq f(y, w, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq y, h(g(y)) \doteq w, y \doteq z$	- EŞEC -

- În ecuația $g(y) \doteq y$, variabila y apare în termenul $g(y)$.
- Nu există unificator pentru aceste ecuații.

Terminarea algoritmului

Propoziție

Algoritmul de unificare se termină.

Terminarea algoritmului

Propoziție

Algoritmul de unificare se termină.

Demonstrație

- Notăm cu
 - N_1 : numărul variabilelor care apar în R
 - N_2 : numărul aparițiilor simbolurilor care apar în R
- Este suficient să arătăm că perechea (N_1, N_2) descrește strict în ordine lexicografică la execuția unui pas al algoritmului:

dacă la execuția unui pas (N_1, N_2) se schimbă în (N'_1, N'_2) , atunci
 $(N_1, N_2) \geq_{lex} (N'_1, N'_2)$

Demonstrație (cont.)

Fiecare regulă a algoritmului modifică N_1 și N_2 astfel:

	N_1	N_2
SCOATE	\geq	$>$
DESCOMPUNE	$=$	$>$
REZOLVĂ	$>$	

- N_1 : numărul variabilelor care apar în R
- N_2 : numărul aparițiilor simbolurilor care apar în R



Unificare în Prolog

- Ce se întâmplă dacă încercăm să unificăm X cu ceva care conține X ?
Exemplu: ?- $X = f(X)$.
- Conform teoriei, acești termeni nu se pot unifica.
- Totuși, multe implementări ale Prolog-ului sără peste această verificare din motive de eficiență.

?- $X = f(X)$.

$X = f(X)$.

Unificare în Prolog

- Ce se întâmplă dacă încercăm să unificăm X cu ceva care conține X ?
Exemplu: `?- X = f(X).`
- Conform teoriei, acești termeni nu se pot unifica.
- Totuși, multe implementări ale Prolog-ului sără peste această verificare din motive de eficiență.

```
?- X = f(X).  
X = f(X).
```

- Putem folosi `unify_with_occurs_check/2`

```
?- unify_with_occurs_check(X,f(X)).  
false.
```

Quiz time!

<https://www.questionpro.com/t/AT4NiZrWmq>



Pe săptămâna viitoare!

Curs 4

2021-2022

Fundamentele Limbajelor de Programare

Cuprins

- 1 Prolog. Liste (continuare)
- 2 Prolog. Tipuri de date compuse
- 3 Planning în Prolog
- 4 Prolog. Reprezentarea unei GIC (optional)
- 5 Prolog. Mai multe despre liste (optional)

Prolog. Liste (continuare)

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: word(relay). word(early). word(layer).

Predicat util:

```
?- name(relay,L). % conversie între atomi și liste
L = [114, 101, 108, 97, 121]
```

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: word(relay). word(early). word(layer).

Predicat util:

```
?- name(relay,L). % conversie între atomi și liste
L = [114, 101, 108, 97, 121]
```

Două abordări posibile:

- se generează o posibilă soluție apoi se testează dacă este în KB.
- se parcurge KB și pentru fiecare termen se testează dacă e soluție.

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: word(relay). word(early). word(layer).

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: word(relay). word(early). word(layer).

```
anagram1(A,B) :- name(A,L), permutation(L,W),
                 name(B,W), word(B).
```

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: word(relay). word(early). word(layer).

```
anagram1(A,B) :- name(A,L), permutation(L,W),  
                 name(B,W), word(B).
```

```
anagram2(A,B) :- name(A,L), word(B),  
                 name(B,W), permutation(L,W).
```

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: word(relay). word(early). word(layer).

```
anagram1(A,B) :- name(A,L), permutation(L,W),  
                 name(B,W), word(B).
```

```
anagram2(A,B) :- name(A,L), word(B),  
                 name(B,W), permutation(L,W).
```

```
?- anagram1(layre,X).  
X = layer ;  
X = relay ;  
X = early ;  
false.
```

```
?- anagram2(layre,X).  
X = relay ;  
X = early ;  
X = layer ;  
false.
```

Recursie

Exercițiu

- Definiți un predicat `rev/2` care verifică dacă o listă este inversa altei liste.

Recursie

Exercițiu

- Definiți un predicat `rev/2` care verifică dacă o listă este inversa altei liste.

```
rev([], []).
```

```
rev([X|T], L) :- rev(T, R), append(R, [X], L).
```

Soluția de mai sus este corectă, dar foarte costisitoare computațional, datorită stilului de programare declarativ.

Cum putem defini o variantă mai rapidă?

O metodă care prin care recursia devine mai rapidă este folosirea **acumulatorilor**, în care se păstrează rezultatele parțiale.

Recursie cu accumulatori

- Varianta inițială:

```
rev([], []).
```

```
rev([X|T], L) :- rev(T, R), append(R, [X], L).
```

- Varianta cu accumulator

```
rev(L, R) :- revac(L, [], R).
```

% la momentul inițial nu am acumulat nimic.

Recursie cu accumulatori

- Varianta inițială:

```
rev([], []).
```

```
rev([X|T], L) :- rev(T, R), append(R, [X], L).
```

- Varianta cu accumulator

```
rev(L, R) :- revac(L, [], R).
```

% la momentul inițial nu am acumulat nimic.

```
revac([], R, R).
```

% cand lista inițială a fost consumată,

% am acumulat rezultatul final.

Recursie cu accumulatori

- Varianta inițială:

```
rev([], []).
```

```
rev([X|T], L) :- rev(T, R), append(R, [X], L).
```

- Varianta cu accumulator

```
rev(L, R) :- revac(L, [], R).
```

% la momentul inițial nu am acumulat nimic.

```
revac([], R, R).
```

% cand lista inițială a fost consumată,

% am acumulat rezultatul final.

```
revac([X|T], Acc, R) :- revac(T, [X|Acc], R).
```

% Acc conține inversa listei care a fost deja parcursă.

- Complexitatea a fost redusă de la $O(n^2)$ la $O(n)$, unde n este lungimea listei.

Prolog. Tipuri de date compuse

Termeni compuși $f(t_1, \dots, t_n)$

- **Termenii** sunt unitățile de bază prin care Prolog reprezintă datele.
- Sunt de 3 tipuri:
 - **Constante**: 23, sansa, 'Jon Snow'
 - **Variabile**: X, Stark, _house
 - **Termeni compuși**:
 - predicate
 - termeni prin care reprezentăm datele

Example

- `born(john, date(20,3,1977))`
 - `born/2` și `date/3` sunt functori
 - `born/2` este un predicat
 - `date/3` definește date compuse

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
 - $[]$ este listă
 - $[X|L]$ este listă, unde X este element și L este listă

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
 - [] este listă
 - [X|L] este listă, unde X este element și L este listă
- Cum definim arborii binari în Prolog?

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
 - [] este listă
 - [X|L] este listă, unde X este element și L este listă
- Cum definim arborii binari în Prolog? Soluție posibilă:

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
 - `[]` este listă
 - `[X|L]` este listă, unde X este element și L este listă
- Cum definim arborii binari în Prolog? Soluție posibilă:
 - `void` este arbore

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
 - `[]` este listă
 - `[X|L]` este listă, unde X este element și L este listă
- Cum definim arborii binari în Prolog? Soluție posibilă:
 - `void` este arbore
 - `tree(X,A1,A2)` este arbore, unde X este un element, iar A1 și A2 sunt arbori

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
 - [] este listă
 - [X|L] este listă, unde X este element și L este listă
- Cum definim arborii binari în Prolog? Soluție posibilă:
 - void este arbore
 - tree(X,A1,A2) este arbore, unde X este un element, iar A1 și A2 sunt arbori

`tree(X,A1,A2)` este un termen compus, dar nu este un predicat!

Arbore binari în Prolog

- Cum arată un arbore?

Arborei binari în Prolog

- Cum arată un arbore?

```
tree(1, tree(2, tree(3, void, void), void), tree(4, void, tree(5, void, void)))
```

Arborei binari în Prolog

- Cum arată un arbore?

tree(1, tree(2, tree(3, void, void), void), tree(4, void, tree(5, void, void)))

- Cum dăm un "nume" arborelui de mai sus?

Arborei binari în Prolog

- Cum arată un arbore?

tree(1, tree(2, tree(3, void, void), void), tree(4, void, tree(5, void, void))))

- Cum dăm un "nume" arborelui de mai sus? Definim un predicat:

```
def(arb, tree(1, tree(2,  
                      tree(3,void,void),  
                      void),  
                      tree(4, void,  
                            tree(5,void,void)))).
```

Arborei binari în Prolog

- Cum arată un arbore?

```
tree(1, tree(2, tree(3, void, void), void), tree(4, void, tree(5, void, void)))
```

- Cum dăm un "nume" arborelui de mai sus? Definim un predicat:

```
def(arb, tree(1, tree(2,  
                      tree(3,void,void),  
                      void),  
                      tree(4, void,  
                            tree(5,void,void)))).
```

Deoarece în Prolog nu avem declarații explicite de date, pentru a defini arborii vom scrie un predicat care este adevărat atunci când argumentul său este un arbore.

Arbore binari în Prolog

Scrieți un predicat care verifică că un termen este arbore binar.

Arborei binari în Prolog

Scrieți un predicat care verifică că un termen este arbore binar.

```
binary_tree(void).  
binary_tree(tree(Element,Left,Right)) :-  
    binary_tree(Left),  
    binary_tree(Right).
```

Arborei binari în Prolog

Scrieti un predicat care verifică că un termen este arbore binar.

```
binary_tree(void).  
binary_tree(tree(Element,Left,Right)) :-  
    binary_tree(Left),  
    binary_tree(Right).
```

Eventual putem defini și un predicat pentru elemente:

```
binary_tree(void).  
binary_tree(tree(Element,Left,Right)) :-  
    binary_tree(Left),  
    binary_tree(Right),  
    element_binary_tree(Element).  
  
element_binary_tree(X) :- integer(X). /* de exemplu */
```

Arborei binari în Prolog

Scrieti un predicat care verifică că un termen este arbore binar.

```
binary_tree(void).  
binary_tree(tree(Element,Left,Right)) :-  
    binary_tree(Left),  
    binary_tree(Right).
```

Eventual putem defini și un predicat pentru elemente:

```
binary_tree(void).  
binary_tree(tree(Element,Left,Right)) :-  
    binary_tree(Left),  
    binary_tree(Right),  
    element_binary_tree(Element).  
  
element_binary_tree(X) :- integer(X). /* de exemplu */  
  
test:- def(arb,T), binary_tree(T).
```

Arborei binari în Prolog

Exercițiu

Scrieti un predicat care verifică dacă un element aparține unui arbore.

Arborei binari în Prolog

Exercițiu

Scrieti un predicat care verifică dacă un element aparține unui arbore.

```
tree_member(X, tree(X,Left,Right)).
```

```
tree_member(X, tree(Y,Left,Right)) :- tree_member(X,Left).
```

```
tree_member(X, tree(Y,Left,Right)) :- tree_member(X,Right).
```

Arborei binari în Prolog

Exercițiu

scrieți un predicat care determină parcurgerea în preordine a unui arbore binar.

Arborei binari în Prolog

Exercițiu

Scrieti un predicat care determină parcurgerea în preordine a unui arbore binar.

```
preorder(tree(X,L,R),Xs) :- preorder(L,Ls),
                                preorder(R,Rs),
                                append([X|Ls],Rs,Xs).

preorder(void,[]).
```

```
test(Tree,Pre):- def(arb, Tree), preorder(Tree,Pre).
```

Arborei binari în Prolog

Exercițiu

Scriți un predicat care determină parcurgerea în preordine a unui arbore binar.

```
preorder(tree(X,L,R),Xs) :- preorder(L,Ls),
                                preorder(R,Rs),
                                append([X|Ls],Rs,Xs).

preorder(void,[]).
```

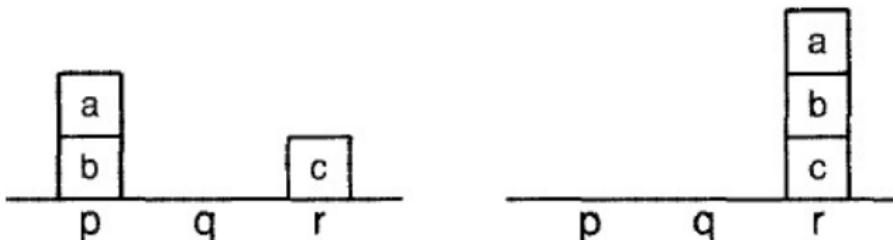
```
test(Tree,Pre):- def(arb, Tree), preorder(Tree,Pre).
```

```
?- test(T,P).
```

```
T = tree(1, tree(2, tree(3, void, void), void), void),
void, tree(5, void, void))),
P = [1, 2, 3, 4, 5]
```

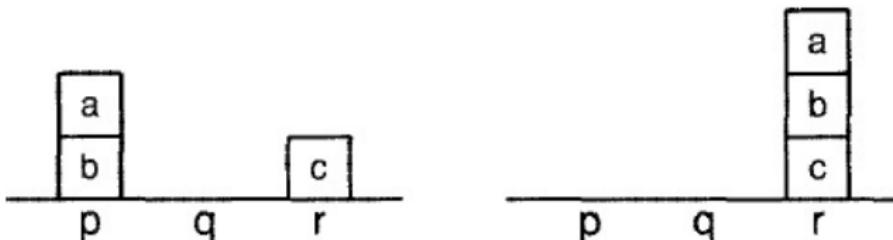
Planning in Prolog

Problemă: Lumea blocurilor



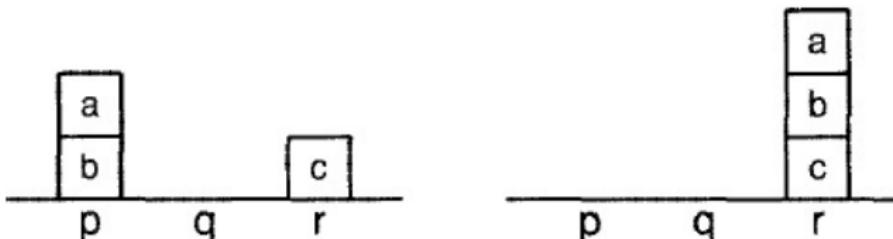
- Lumea blocurilor este formată din:
 - trei blocuri: a,b, c
 - trei poziții: p,q, r
 - un bloc poate sta peste un alt bloc sau pe o poziție

Problemă: Lumea blocurilor



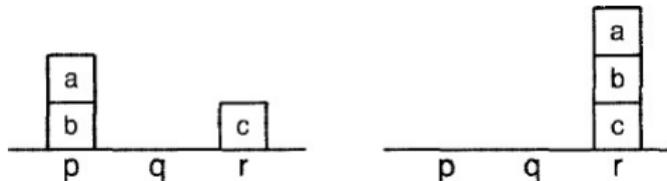
- Lumea blocurilor este formată din:
 - trei blocuri: a,b, c
 - trei poziții: p,q, r
 - un bloc poate sta peste un alt bloc sau pe o poziție
- Un bloc poate fi mutat pe o poziție liberă sau pe un alt bloc.

Problemă: Lumea blocurilor



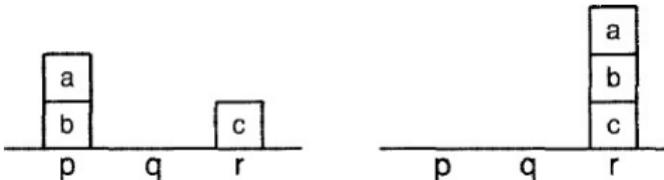
- Lumea blocurilor este formată din:
 - trei blocuri: a,b, c
 - trei poziții: p,q, r
 - un bloc poate sta peste un alt bloc sau pe o poziție
- Un bloc poate fi mutat pe o poziție liberă sau pe un alt bloc.
- Problema este de a găsi un sir de mutări astfel încât dintr-o stare inițială să se ajungă într-o stare finală

Lumea blocurilor



- Reprezentarea blocurilor, pozițiilor și a stărilor:
block(a). block(b). block(c).
place(p). place(q). place(r).

Lumea blocurilor

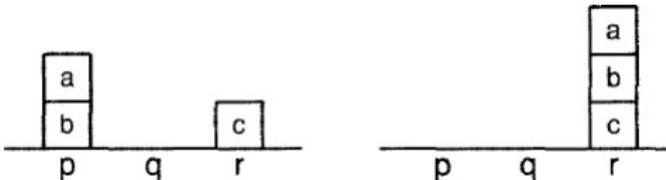


□ Reprezentarea blocurilor, pozițiilor și a stărilor:

block(a). block(b). block(c).
place(p). place(q). place(r).

```
initial_state([on(a,b), on(b,p), on(c,r)]).  
final_state([on(a,b), on(b,c), on(c,r)]).
```

Lumea blocurilor



□ Reprezentarea blocurilor, pozițiilor și a stărilor:

`block(a).` `block(b).` `block(c).`

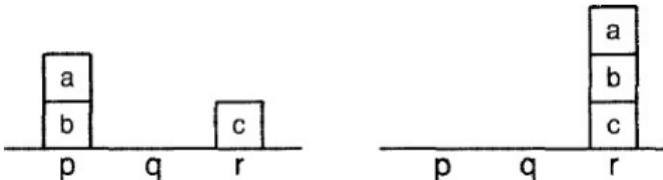
`place(p).` `place(q).` `place(r).`

`initial_state([on(a,b), on(b,p), on(c,r)]).`

`final_state([on(a,b), on(b,c), on(c,r)]).`

Observați că `on(a,b)`, `on(b,c)`, etc. sunt date compuse.

Lumea blocurilor



- Reprezentarea blocurilor, pozițiilor și a stărilor:

`block(a).` `block(b).` `block(c).`

`place(p).` `place(q).` `place(r).`

`initial_state([on(a,b), on(b,p), on(c,r)]).`

`final_state([on(a,b), on(b,c), on(c,r)]).`

Observați că `on(a,b)`, `on(b,c)`, etc. sunt date compuse.

- O **stare** este o listă de termenii de tipul `on(X,Y)`.

Într-o listă care reprezintă o stare, termenii `on(X,Y)` sunt ordonați după prima componentă.

Lumea blocurilor

- Predicatul `valid_plan(State1, State2, Plan)` va **genera** în variabila `Plan` un sir de mutări permise care transformă starea `State1` în starea `State2`.

```
valid_plan(State1, State2, Plan) :-  
    valid_plan_aux(State1, State2, [State1], Plan).
```

Lumea blocurilor

- Predicatul `valid_plan(State1, State2, Plan)` va **genera** în variabila Plan un sir de mutări permise care transformă starea State1 în starea State2.

```
valid_plan(State1, State2, Plan) :-  
    valid_plan_aux(State1, State2, [State1], Plan).  
  
valid_plan_aux(State, State, _, []).  
  
valid_plan_aux(State1, State2, Visited, [Action|Actions]) :-  
    legal_action(Action, State1),  
    update(Action, State1, State),  
    \+ member(State, Visited),  
    valid_plan_aux(State, State2, [State|Visited], Actions).
```

- În modelarea noastră, `valid_plan_aux/4` este un predicat auxiliar, cu ajutorul căruia reținem stările "vizitate".
- Căutare de tip **depth-first**.

Lumea blocurilor

- Predicatul `legal_action(Action, State)` va instanția `Action` cu o mutare care poate fi efectuată în starea `State`. Există două mutări posibile: mutarea pe un bloc și mutarea pe o poziție.

Lumea blocurilor

- Predicatul `legal_action(Action, State)` va instanția Action cu o mutare care poate fi efectuată în starea State. Există două mutări posibile: mutarea pe un bloc și mutarea pe o poziție.

```
clear(X,State) :- \+ member(on(_,X),State).
```

Lumea blocurilor

- Predicatul `legal_action(Action, State)` va instanția Action cu o mutare care poate fi efectuată în starea State. Există două mutări posibile: mutarea pe un bloc și mutarea pe o poziție.

```
clear(X,State) :- \+ member(on(_,X),State).  
  
legal_action(to_block(Block1,Block2),State) :-  
    block(Block1), clear(Block1,State),  
    block(Block2), Block1 \== Block2,  
    clear(Block2,State).
```

Lumea blocurilor

- Predicatul `legal_action(Action, State)` va instanția Action cu o mutare care poate fi efectuată în starea State. Există două mutări posibile: mutarea pe un bloc și mutarea pe o poziție.

```
clear(X,State) :- \+ member(on(_,X),State).  
  
legal_action(to_block(Block1,Block2),State) :-  
    block(Block1), clear(Block1,State),  
    block(Block2), Block1 \== Block2,  
    clear(Block2,State).  
  
legal_action(to_place(Block,Place),State) :-  
    block(Block), clear(Block,State),  
    place(Place), clear(Place,State).
```

Lumea blocurilor

- Predicatul `update(Action, State, State1)` are următoarea semnificație: făcând mutarea `Action` în starea `State` se ajunge în starea `State1`.

Lumea blocurilor

- Predicatul `update(Action, State, State1)` are următoarea semnificație: făcând mutarea `Action` în starea `State` se ajunge în starea `State1`.

```
update(to_block(X,Z), State, State1) :-  
    substitute(on(X,_), on(X,Z), State, State1).
```

```
update(to_place(X,Z), State, State1) :-  
    substitute(on(X,_), on(X,Z), State, State1).
```

Lumea blocurilor

- Predicatul `update(Action, State, State1)` are următoarea semnificație: făcând mutarea Action în starea State se ajunge în starea State1.

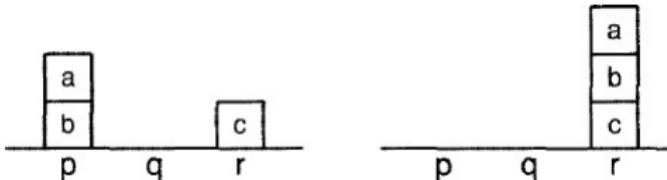
```
update(to_block(X,Z),State,State1) :-  
    substitute(on(X,_),on(X,Z),State,State1).
```

```
update(to_place(X,Z),State,State1) :-  
    substitute(on(X,_),on(X,Z),State,State1).
```

- `substitute(X, Y, L, R)` substituie X cu Y în lista L, rezultatul fiind R.

```
substitute(X,Y,[X|Xs],[Y|Xs]).  
substitute(X,Y,[X1|Xs],[X1|Ys]) :- X \== X1,  
    substitute(X,Y,Xs,Ys).
```

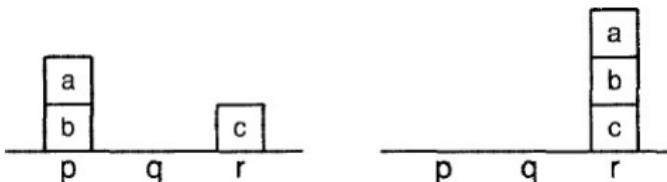
Lumea blocurilor



block(a). block(b). block(c).
place(p). place(q). place(r).

initial_state([on(a,b), on(b,p), on(c,r)]).
final_state([on(a,b), on(b,c), on(c,r)]).

Lumea blocurilor



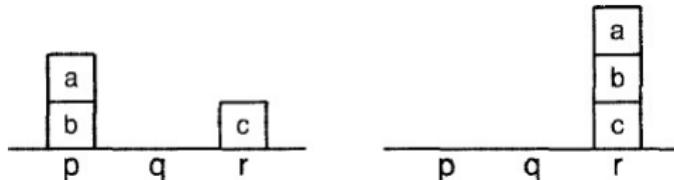
```
block(a). block(b). block(c).  
place(p). place(q). place(r).
```

```
initial_state([on(a,b) , on(b,p) , on(c,r)]).  
final_state([on(a,b) , on(b,c) , on(c,r)]).
```

```
test_plan(Plan) :- initial_state(I) , final_state(F) ,  
valid_plan(I,F,Plan).
```

```
?- test(Plan).
```

Lumea blocurilor



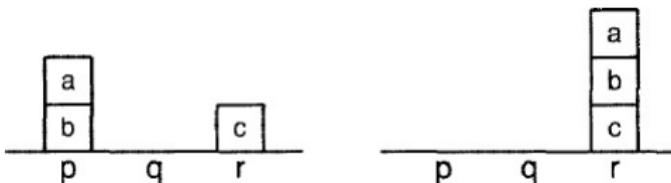
```
test_plan(Plan) :- initial_state(I), final_state(F),
                  valid_plan(I,F,Plan).
?- test(Plan).
Plan = [to_block(a, c), to_block(b, a), to_place(b, q),
        to_block(a, b), to_block(c, a), to_place(c, p),
        to_block(a, c), to_block(b, a), to_place(b, r),
        to_block(a, b), to_block(c, a), to_place(c, q),
        to_block(a, c), to_block(b, a), to_place(b, p),
        to_block(a, b), to_place(a, r), to_block(b, a),
        to_block(b, c), to_block(a, b), to_place(a, p),
        to_block(b, a), to_block(c, b), to_place(c, r),
        to_block(b, c), to_block(a, b)]
```

Lumea blocurilor

Pentru a obține o soluție mai simplă, putem limita numărul de mutări!

```
valid_plan(State1, State2, Plan, N) :-  
    valid_plan_aux(State1, State2, [State1], Plan, N).  
  
valid_plan_aux(State, State, _, [], _).  
  
valid_plan_aux(State1, State2, Visited, [Action|Actions], N) :-  
    legal_action(Action, State1),  
    update(Action, State1, State),  
    \+ member(State, Visited), length(Visited, M), M < N,  
    valid_plan_aux(State, State2, [State|Visited], Actions, N).
```

Lumea blocurilor



```
test_plan(Plan,N) :- initial_state(I), final_state(F),
                     valid_plan(I,F,Plan,N).
```

```
?- test(Plan,3).
```

false

```
?- test(Plan,4).
```

```
Plan = [to_place(a, q), to_block(b, c), to_block(a, b)]
```

În general

- Predicatul `valid_plan(State1, State2, Plan)` generează, printr-o căutare de tip depth-first, în variabila `Plan` un sir de mutări permise care transformă starea `State1` în starea `State2`.

```
valid_plan(State1, State2, Plan) :-  
    valid_plan_aux(State1, State2, [State1], Plan).  
  
valid_plan_aux(State, State, _, []).  
  
valid_plan_aux(State1, State2, Visited, [Action|Actions]) :-  
    legal_action(Action, State1),  
    update(Action, State1, State),  
    \+ member(State, Visited),  
    valid_plan_aux(State, State2, [State|Visited], Actions).
```

- Reprezentarea stărilor, a acțiunilor, a soluției depinde de problema concretă pe care o rezolvăm.

Prolog. Reprezentarea unei GIC (optional)

Structura frazelor

- Aristotel, On Interpretation,

<http://classics.mit.edu/Aristotle/interpretation.1.1.html>:

"Every affirmation, then, and every denial, will consist of a noun and a verb, either definite or indefinite."

Structura frazelor

- Aristotel, On Interpretation,
<http://classics.mit.edu/Aristotle/interpretation.1.1.html>:
"Every affirmation, then, and every denial, will consist of a noun and a verb, either definite or indefinite."
- N. Chomsky, Syntactic structure, Mouton Publishers, First printing 1957 - Fourteenth printing 1985 [Chapter 4 (Phrase Structure)]
 - (i) *Sentence* → *NP + VP*
 - (ii) *NP* → *T + N*
 - (iii) *VP* → *Verb + NP*
 - (iv) *T* → *the*
 - (v) *N* → *man, ball, etc.*
 - (vi) *V* → *hit, took, etc.*

Gramatică independentă de context

- Definim structura propozițiilor folosind o gramatică independentă de context:

S	→	NP VP
NP	→	Det N
VP	→	V
VP	→	V NP
Det	→	<i>the</i>
Det	→	<i>a</i>
N	→	<i>boy</i>
N	→	<i>girl</i>
V	→	<i>loves</i>
V	→	<i>hates</i>

- Neterminalele definesc categorii gramaticale:
 - S (propozițiile),
 - NP (expresiile substantivale),
 - VP (expresiile verbale),
 - V (verbele),
 - N (substantivele),
 - Det (articolele).
- Terminalele definesc cuvintele.

Gramatică independentă de context

GIC

S	\rightarrow	NP VP	Det	\rightarrow	<i>the</i>
NP	\rightarrow	Det N	Det	\rightarrow	<i>a</i>
VP	\rightarrow	V	N	\rightarrow	<i>boy</i>
VP	\rightarrow	V NP	N	\rightarrow	<i>girl</i>
			V	\rightarrow	<i>loves</i>
			V	\rightarrow	<i>hates</i>

Ce vrem să facem?

- Vrem să scriem un program în Prolog care să recunoască propozițiile generate de această gramatică.
- Reprezentăm propozițiile prin liste.

```
?- atomic_list_concat(SL, ' ', 'a boy loves a girl').  
SL = [a, boy, loves, a, girl]
```

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

SL = [a, boy, loves, a, girl]

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

SL = [a, boy, loves, a, girl]

- Fiecăruia neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective.

n([boy]). n([girl]). det([the]). v([loves]).

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

SL = [a, boy, loves, a, girl]

- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective.

n([boy]). n([girl]). det([the]). v([loves]).

- Lista asociată unei propoziții se obține prin concatenarea listelor asociate elementelor componente.

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.
SL = [a, boy, loves, a, girl]
- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective.
n([boy]). n([girl]). det([the]). v([loves]).
- Lista asociată unei propoziții se obține prin concatenarea listelor asociate elementelor componente.

De exemplu, interpretăm regula $S \rightarrow NP\ VP$ astfel:

o propoziție este o listă L care se obține prin concatenarea a două liste, X și Y, unde X reprezintă o expresie substantivală și Y reprezintă o expresie verbală.

`s(L) :- np(X), vp(Y), append(X, Y, L).`

Definirea unei gramatici în Prolog

Gramatică independentă de context

S	→	NP VP	Det	→	<i>the</i>
NP	→	Det N	Det	→	<i>a</i>
VP	→	V	N	→	<i>boy</i>
VP	→	V NP	N	→	<i>girl</i>
			V	→	<i>loves</i>
			V	→	<i>hates</i>

Prolog

```
s(L) :- np(X), vp(Y),
        append(X, Y, L).
np(L) :- det(X), n(Y),
        append(X, Y, L).
vp(L) :- v(L).
vp(L) :- v(X), np(Y),
        append(X, Y, L).

det([the]). 
det([a]). 
n([boy]). 
n([girl]). 
v([loves]). 
v([hates]).
```

Definirea unei gramatici în Prolog

```
s(L) :- np(X), vp(Y),  
        append(X, Y, L).  
  
np(L) :- det(X), n(Y),  
        append(X, Y, L).  
  
vp(L) :- v(L).  
vp(L) :- v(X), np(Y),  
        append(X, Y, L).  
  
det([the]).  
det([a]).  
n([boy]).  
n([girl]).  
v([loves]).  
v([hates]).  
  
?- s([a,boy,loves,a,girl]).  
true .  
  
?- s[a,girl|T].  
T = [loves] ;  
T = [hates] ;  
T = [loves, the, boy] ;  
:  
:  
?- s(S).  
S = [the, boy, loves] ;  
S = [the, boy, hates] ;  
:  
:
```

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.
SL = [a, boy, loves, a, girl]
- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective.
`n([boy]). n([girl]). det([the]). v([loves]).`
- Lista asociată unei propoziții se obține prin concatenarea listelor asociate elementelor componente.
- Deși corectă, reprezentarea anterioară este ineficientă, arborele de căutare este foarte mare. Pentru a optimiza, folosim reprezentarea listelor ca diferențe.

Prolog. Mai multe despre liste (optional)

Liste append/3

- Reamintim definiția funcției append/3:

```
?- listing	append/3 .  
append([], L, L).  
append([X|T], L, [X|R]) :- append(T, L, R).
```

```
?- append(X, Y, [a, b, c]).
```

```
X = [],
```

```
Y = [a, b, c] ;
```

```
X = [a],
```

```
Y = [b, c] ;
```

```
X = [a, b],
```

```
Y = [c] ;
```

```
X = [a, b, c],
```

```
Y = [] ;
```

false

- Funcția astfel definită poate fi folosită atât pentru verificare, cât și pentru generare.

Liste

```
append([],L,L).  
append([X|T],L, [X|R]) :- append(T,L,R).
```

Exercițiu

Definiți `prefix/2` și `suffix/2` folosind `append`.

Liste

```
append([],L,L).  
append([X|T],L, [X|R]) :- append(T,L,R).
```

Exercițiu

Definiți **prefix/2** și **suffix/2** folosind append.

```
prefix(P,L) :- append(P,_, L).  
suffix(S,L) :- append( _,S,L).
```

Liste

```
append([],L,L).  
append([X|T],L, [X|R]) :- append(T,L,R).
```

Exercițiu

Definiți `prefix/2` și `suffix/2` folosind `append`.

```
prefix(P,L) :- append(P,_, L).  
suffix(S,L) :- append(_,S,L).
```

Observăm că funcția `append` parcurge prima listă.

Am putea rescrie această funcție astfel încât legătura să se facă direct, aşa cum putem face în programarea imperativă?

Liste

```
append([], L, L).  
append([X|T], L, [X|R]) :- append(T, L, R).
```

Exercițiu

Definiți **prefix/2** și **suffix/2** folosind append.

```
prefix(P, L) :- append(P, _, L).  
suffix(S, L) :- append(_, S, L).
```

Observăm că funcția append parcurge prima listă.

Am putea rescrie această funcție astfel încât legătura să se facă direct, aşa cum putem face în programarea imperativă?

Problema poate fi rezolvată scriind **listele ca diferențe**, o tehnică utilă în limbajul Prolog.

Liste ca diferențe

- Ideea: lista $[t_1, \dots, t_n]$ va fi reprezentată printr-o pereche $([t_1, \dots, t_n] | T, T)$
- Această pereche poate fi notată $[t_1, \dots, t_n | T] - T$, dar notația nu este importantă.

Liste ca diferențe

- Ideea: lista $[t_1, \dots, t_n]$ va fi reprezentată printr-o pereche $([t_1, \dots, t_n | T], T)$

Această pereche poate fi notată $[t_1, \dots, t_n | T] - T$, dar notația nu este importantă.

- Vrem să definim append/3 pentru liste ca diferențe:

`dlappend((X1,T1),(X2,T2),(R,T)) :- ?.`

?- `dlappend(([1,2,3|P],P),([4,5|T],T),RD).`

P = [4, 5|T],

RD = ([1, 2, 3, 4, 5|T], T).

Liste ca diferențe ([t₁, ..., t_n|T], T)

```
dlappend((X1,T1),(X2,T2),(R,T)) :- ?.
```

Liste ca diferențe ([t₁, ..., t_n] | T), T

dlappend((X₁,T₁),(X₂,T₂),(R,T)) :- ?.

- Dacă [t₁, ..., t_n] este diferența (X₁,T₁), iar [q₁, ..., q_k] este diferența (X₂,T₂) observăm că diferența (R,T) trebuie să fie [t₁, ..., t_n, q₁, ..., q_k].

Liste ca diferențe ([t₁, ..., t_n|T], T)

dlappend((X₁,T₁),(X₂,T₂),(R,T)) :- ?.

- Dacă [t₁, ..., t_n] este diferența (X₁,T₁), iar [q₁, ..., q_k] este diferența (X₂,T₂) observăm că diferența (R,T) trebuie să fie [t₁, ..., t_n, q₁..., q_k].
- Obținem R=[t₁, ..., t_n, q₁..., q_k|T], deci
 $(X_1, T_1) = (R, P)$ și $(X_2, T_2) = (P, T)$
unde $P = [q_1, \dots, q_k | T]$.

Liste ca diferențe ([t₁, ..., t_n|T], T)

dlappend((X₁, T₁), (X₂, T₂), (R, T)) :- ?.

- Dacă [t₁, ..., t_n] este diferența (X₁, T₁), iar [q₁, ..., q_k] este diferența (X₂, T₂) observăm că diferența (R, T) trebuie să fie [t₁, ..., t_n, q₁..., q_k].
- Obținem R=[t₁, ..., t_n, q₁..., q_k|T], deci (X₁, T₁) = (R, P) și (X₂, T₂) = (P, T) unde P =[q₁, ..., q_k|T]).
- Definiția este:

dlappend((R,P),(P,T),(R,T)).

?- dlappend(([1,2,3|P],P),([4,5|T],T),RD).

P = [4, 5|T],

RD = ([1, 2, 3, 4, 5|T], T).

Liste ca diferențe ([t₁, ..., t_n|T], T)

dlappend((X₁, T₁), (X₂, T₂), (R, T)) :- ?.

- Dacă [t₁, ..., t_n] este diferența (X₁, T₁), iar [q₁, ..., q_k] este diferența (X₂, T₂) observăm că diferența (R, T) trebuie să fie [t₁, ..., t_n, q₁..., q_k].
- Obținem R=[t₁, ..., t_n, q₁..., q_k|T], deci (X₁, T₁) = (R, P) și (X₂, T₂) = (P, T) unde P =[q₁, ..., q_k|T]).
- Definiția este:

dlappend((R,P),(P,T),(R,T)).

?- dlappend(([1,2,3|P],P),([4,5|T],T),RD).

P = [4, 5|T],

RD = ([1, 2, 3, 4, 5|T], T).

- dlappend este foarte rapid, dar nu poate fi folosit pentru generare, ci numai pentru verificare.

Recursie

- Multe implementări ale limbajului Prolog aplică "last call optimization" atunci când un apel recursiv este ultimul predicat din corpul unei clauze (tail recursion).
- Atunci când este posibil, se recomandă utilizare recursiei la coadă (tail recursion).
- Vom defini un predicat care generează liste lungi în două moduri și vom analiza performanța folosind predicatul `time/1`.

Recursie

- Multe implementări ale limbajului Prolog aplică "last call optimization" atunci când un apel recursiv este ultimul predicat din corpul unei clauze (tail recursion).
- Atunci când este posibil, se recomandă utilizare recursiei la coadă (tail recursion).
- Vom defini un predicat care generează liste lungi în două moduri și vom analiza performanța folosind predicatul **time/1**.

```
biglist([]).
```

```
biglist([N|T]) :- N >= 1, M is N-1, biglist(M,T), M=M.
```

```
biglist_tr([]).
```

```
biglist_tr([N|T]) :- N >= 1, M is N-1, biglist_tr(M,T).
```

Recursie la coadă

- Predicat **fără** recursie la coadă:

```
biglist([],[]).
```

```
biglist([N|T]) :- N >= 1, M is N-1, biglist(M,T), M=M.
```

Apelul recursiv întoarce valoarea găsită în predicatul apelant, acestă valoare urmând a fi prelucrată.

Recursie la coadă

- Predicat **fără** recursie la coadă:

```
biglist([],[]).
```

```
biglist([N|T]) :- N >= 1, M is N-1, biglist(M,T), M=M.
```

Apelul recursiv întoarce valoarea găsită în predicatul apelant, acestă valoare urmând a fi prelucrată.

```
?- time(biglist(50000,X)).
```

```
100,000 inferences, 0.016 CPU in 0.038 seconds  
(41% CPU, 6400000 Lips)
```

```
X = [50000, 49999, 49998|...].
```

Recursie la coadă

- Predicat **fără** recursie la coadă:

```
biglist([],[]).
```

```
biglist([N|T]) :- N >= 1, M is N-1, biglist(M,T), M=M.
```

Apelul recursiv întoarce valoarea găsită în predicatul apelant, acestă valoare urmând a fi prelucrată.

```
?- time(biglist(50000,X)).
```

```
100,000 inferences, 0.016 CPU in 0.038 seconds  
(41% CPU, 6400000 Lips)
```

```
X = [50000, 49999, 49998|...].
```

- Predicatul **cu** recursie la coadă:

```
biglist_tr([],[]).
```

```
biglist_tr([N|T]) :- N >= 1, M is N-1, biglist_tr(M,T).
```

Recursie la coadă

- Predicat **fără** recursie la coadă:

```
biglist([],[]).
```

```
biglist([N|T]) :- N >= 1, M is N-1, biglist(M,T), M=M.
```

Apelul recursiv întoarce valoarea găsită în predicatul apelant, acestă valoare urmând a fi prelucrată.

```
?- time(biglist(50000,X)).
```

```
100,000 inferences, 0.016 CPU in 0.038 seconds  
(41% CPU, 6400000 Lips)
```

```
X = [50000, 49999, 49998|...].
```

- Predicatul **cu** recursie la coadă:

```
biglist_tr([],[]).
```

```
biglist_tr([N|T]) :- N >= 1, M is N-1, biglist_tr(M,T).
```

```
?- time(biglist_tr(50000,X)).
```

```
100,000 inferences, 0.000 CPU in 0.007 seconds  
(0% CPU, Infinite Lips)
```

```
X = [50000, 49999, 49998|...]
```

Pe săptămâna viitoare!

Curs 5

2021-2022

Fundamentele Limbajelor de Programare

Cuprins

- 1 Logica propozițională PL**
- 2 PL - Deducreție naturală**

Logica propozițională PL

Logica propozițională PL

- O propoziție este un enunț care poate fi adevărat (1) sau fals (0).
- Propozitiile sunt notate simbolic ($\varphi, \psi, \chi, \dots$) și sunt combinate cu ajutorul conectorilor logici ($\neg, \rightarrow, \vee, \wedge, \leftrightarrow$).

Logica propozițională PL

- O propoziție este un enunț care poate fi adevărat (1) sau fals (0).
- Propozitiile sunt notate simbolic ($\varphi, \psi, \chi, \dots$) și sunt combinate cu ajutorul conectorilor logici ($\neg, \rightarrow, \vee, \wedge, \leftrightarrow$).

Example

Fie φ propoziția:

$$(\text{stark} \wedge \neg \text{dead}) \rightarrow (\text{sansa} \vee \text{arya} \vee \text{bran})$$

Logica propozițională PL

- O propoziție este un enunț care poate fi adevărat (1) sau fals (0).
- Propozitiile sunt notate simbolic ($\varphi, \psi, \chi, \dots$) și sunt combinate cu ajutorul conectorilor logici ($\neg, \rightarrow, \vee, \wedge, \leftrightarrow$).

Example

Fie φ propoziția:

$$(\text{stark} \wedge \neg \text{dead}) \rightarrow (\text{sansa} \vee \text{arya} \vee \text{bran})$$

Cine este $\neg\varphi$?

Logica propozițională PL

- O propoziție este un enunț care poate fi adevărat (1) sau fals (0).
- Propozitiile sunt notate simbolic ($\varphi, \psi, \chi, \dots$) și sunt combinate cu ajutorul conectorilor logici ($\neg, \rightarrow, \vee, \wedge, \leftrightarrow$).

Example

Fie φ propoziția:

$$(\text{stark} \wedge \neg \text{dead}) \rightarrow (\text{sansa} \vee \text{arya} \vee \text{bran})$$

Cine este $\neg\varphi$? Propoziția $\neg\varphi$ este:

$$\text{stark} \wedge \neg \text{dead} \wedge \neg \text{sansa} \wedge \neg \text{arya} \wedge \neg \text{bran}$$

Limbajul și formulele PL

□ Limbajul PL

- variabile propoziționale: $\text{Var} = \{p, q, v, \dots\}$
- conectori logici: \neg (unar), \rightarrow , \wedge , \vee , \leftrightarrow (binari)

□ Formulele PL

$\text{var} ::= p \mid q \mid v \mid \dots$

$\text{form} ::= \text{var} \mid (\neg \text{form}) \mid \text{form} \wedge \text{form} \mid \text{form} \vee \text{form}$
 $\mid \text{form} \rightarrow \text{form} \mid \text{form} \leftrightarrow \text{form}$

Limbajul și formulele PL

□ Limbajul PL

- variabile propoziționale: $\text{Var} = \{p, q, v, \dots\}$
- conectori logici: \neg (unar), \rightarrow , \wedge , \vee , \leftrightarrow (binari)

□ Formulele PL

$\text{var} ::= p \mid q \mid v \mid \dots$
 $\text{form} ::= \text{var} \mid (\neg \text{form}) \mid \text{form} \wedge \text{form} \mid \text{form} \vee \text{form}$
 $\mid \text{form} \rightarrow \text{form} \mid \text{form} \leftrightarrow \text{form}$

Example

- Nu sunt formule: $v_1 \neg \rightarrow (v_2)$, $\neg v_1 v_2$
- Sunt formule: $((v_1 \rightarrow v_2) \rightarrow (\neg v_1))$, $(\neg(v_1 \rightarrow v_2))$

Limbajul și formulele PL

□ Limbajul PL

- variabile propoziționale: $\text{Var} = \{p, q, v, \dots\}$
- conectori logici: \neg (unar), \rightarrow , \wedge , \vee , \leftrightarrow (binari)

□ Formulele PL

$$\begin{aligned}\text{var} &::= p \mid q \mid v \mid \dots \\ \text{form} &::= \text{var} \mid (\neg \text{form}) \mid \text{form} \wedge \text{form} \mid \text{form} \vee \text{form} \\ &\quad \mid \text{form} \rightarrow \text{form} \mid \text{form} \leftrightarrow \text{form}\end{aligned}$$

Example

- Nu sunt formule: $v_1 \neg \rightarrow (v_2)$, $\neg v_1 v_2$
 - Sunt formule: $((v_1 \rightarrow v_2) \rightarrow (\neg v_1))$, $(\neg(v_1 \rightarrow v_2))$
- Notăm cu *Form* mulțimea formulelor.

Limbajul și formulele PL

□ Limbajul PL

- variabile propoziționale: $\text{Var} = \{p, q, v, \dots\}$
- conectori logici: \neg (unar), \rightarrow , \wedge , \vee , \leftrightarrow (binari)

□ Formulele PL

$\text{var} ::= p \mid q \mid v \mid \dots$

$\text{form} ::= \text{var} \mid (\neg \text{form}) \mid \text{form} \wedge \text{form} \mid \text{form} \vee \text{form}$
 $\mid \text{form} \rightarrow \text{form} \mid \text{form} \leftrightarrow \text{form}$

- Conectorii sunt împărțiți în conectori **de bază** și conectori **derivați** (în funcție de formalism).
- Legături între conectori:

$$\varphi \vee \psi := \neg \varphi \rightarrow \psi$$

$$\varphi \wedge \psi := \neg(\varphi \rightarrow \neg \psi)$$

$$\varphi \leftrightarrow \psi := (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$$

Sintaxa și semantica

Un sistem logic are două componente:

- Sintaxa

- Semantica

Sintaxa și semantica

Un sistem logic are două componente:

- **Sintaxa**

- noțiuni sintactice: demonstrație, teoremă
- notăm prin $\vdash \varphi$ faptul că φ este teoremă
- notăm prin $\Gamma \vdash \varphi$ faptul că formula φ este demonstrabilă din mulțimea de formule Γ

- **Semantica**

Sintaxa și semantica

Un sistem logic are două componente:

□ Sintaxa

- noțiuni sintactice: demonstrație, teoremă
- notăm prin $\vdash \varphi$ faptul că φ este teoremă
- notăm prin $\Gamma \vdash \varphi$ faptul că formula φ este demonstrabilă din mulțimea de formule Γ

□ Semantica

- noțiuni semantice: adevăr, model, tautologie (formulă universal adevărată)
- notăm prin $\models \varphi$ faptul că φ este tautologie
- notăm prin $\Gamma \models \varphi$ faptul că formula φ este adevărată atunci când toate formulele din mulțimea Γ sunt adevărate

Logica propozițională

Example

Formalizați următorul raționament:

If winter is coming and Ned is not alive then Robb is lord of Winterfell. Winter is coming. Rob is not lord of Winterfell. Then Ned is alive.

Logica propozițională

Example

Formalizați următorul raționament:

If winter is coming and Ned is not alive then Robb is lord of Winterfell. Winter is coming. Rob is not lord of Winterfell. Then Ned is alive.

O posibilă formalizare este următoarea:

p = winter is coming

q = Ned is alive

r = Robb is lord of Winterfel

Logica propozițională

Example

Formalizați următorul raționament:

If winter is coming and Ned is not alive then Robb is lord of Winterfell. Winter is coming. Rob is not lord of Winterfell. Then Ned is alive.

O posibilă formalizare este următoarea:

p = winter is coming

q = Ned is alive

r = Robb is lord of Winterfel

$$\{(p \wedge \neg q) \rightarrow r, p, \neg r\} \models q$$

Semantica PL

- Mulțimea valorilor de adevăr este $\{0, 1\}$ pe care considerăm următoarele operații:

x	$\neg x$
0	1
1	0

x	y	$x \rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

$$x \vee y := \max\{x, y\}$$

$$x \wedge y := \min\{x, y\}$$

Semantica PL

- o funcție $e : \text{Var} \rightarrow \{0, 1\}$ se numește evaluare (interpretare)

Semantica PL

- o funcție $e : \text{Var} \rightarrow \{0, 1\}$ se numește **evaluare** (interpretare)
- pentru orice evaluare $e : \text{Var} \rightarrow \{0, 1\}$ există o unică funcție $e^+ : \text{Form} \rightarrow \{0, 1\}$ care verifică următoarele proprietăți:
 - $e^+(v) = e(v)$
 - $e^+(\neg\varphi) = \neg e^+(\varphi)$
 - $e^+(\varphi \rightarrow \psi) = e^+(\varphi) \rightarrow e^+(\psi)$
 - $e^+(\varphi \wedge \psi) = e^+(\varphi) \wedge e^+(\psi)$
 - $e^+(\varphi \vee \psi) = e^+(\varphi) \vee e^+(\psi)$

oricare ar fi $v \in \text{Var}$ și $\varphi, \psi \in \text{Form}$.

Semantica PL

- o funcție $e : \text{Var} \rightarrow \{0, 1\}$ se numește **evaluare** (interpretare)
- pentru orice evaluare $e : \text{Var} \rightarrow \{0, 1\}$ există o unică funcție $e^+ : \text{Form} \rightarrow \{0, 1\}$ care verifică următoarele proprietăți:
 - $e^+(v) = e(v)$
 - $e^+(\neg\varphi) = \neg e^+(\varphi)$
 - $e^+(\varphi \rightarrow \psi) = e^+(\varphi) \rightarrow e^+(\psi)$
 - $e^+(\varphi \wedge \psi) = e^+(\varphi) \wedge e^+(\psi)$
 - $e^+(\varphi \vee \psi) = e^+(\varphi) \vee e^+(\psi)$

oricare ar fi $v \in \text{Var}$ și $\varphi, \psi \in \text{Form}$.

Example

Dacă $e(p) = 0$ și $e(q) = 1$ atunci

$$e^+(p \vee (p \rightarrow q)) = e^+(p) \vee e^+(p \rightarrow q) = e(p) \vee (e(p) \rightarrow e(q)) = 1$$

Semantica PL

Considerăm $\Gamma \cup \{\varphi\} \subseteq Form.$

Semantica PL

Considerăm $\Gamma \cup \{\varphi\} \subseteq Form$.

- O evaluare $e : Var \rightarrow \{0, 1\}$ este **model** al formulei φ dacă $e^+(\varphi) = 1$.
Evaluarea e este **model** al lui Γ dacă $e^+(\Gamma) = \{1\}$, i.e. $e^+(\gamma) = 1$ oricare $\gamma \in \Gamma$.

Semantica PL

Considerăm $\Gamma \cup \{\varphi\} \subseteq Form$.

- O evaluare $e : Var \rightarrow \{0, 1\}$ este **model** al formulei φ dacă $e^+(\varphi) = 1$.
Evaluarea e este **model** al lui Γ dacă $e^+(\Gamma) = \{1\}$, i.e. $e^+(\gamma) = 1$ oricare $\gamma \in \Gamma$.
- O formulă φ este **satisfiabilă** dacă are un model. O mulțime Γ de formule este **satisfiabilă** dacă are un model.

Semantica PL

Considerăm $\Gamma \cup \{\varphi\} \subseteq Form$.

- O evaluare $e : Var \rightarrow \{0, 1\}$ este **model** al formulei φ dacă $e^+(\varphi) = 1$.
Evaluarea e este **model** al lui Γ dacă $e^+(\Gamma) = \{1\}$, i.e. $e^+(\gamma) = 1$ oricare $\gamma \in \Gamma$.
- O formulă φ este **satisfiabilă** dacă are un model. O mulțime Γ de formule este **satisfiabilă** dacă are un model.
- O formulă φ este **tautologie** (**validă, universal adevarată**) dacă $e^+(\varphi) = 1$ pentru orice evaluare $e : Var \rightarrow \{0, 1\}$.
Notăm prin $\models \varphi$ faptul că φ este o tautologie.

Semantica PL

Considerăm $\Gamma \cup \{\varphi\} \subseteq Form$.

- O evaluare $e : Var \rightarrow \{0, 1\}$ este **model** al formulei φ dacă $e^+(\varphi) = 1$.
Evaluarea e este **model** al lui Γ dacă $e^+(\Gamma) = \{1\}$, i.e. $e^+(\gamma) = 1$ oricare $\gamma \in \Gamma$.
- O formulă φ este **satisfiabilă** dacă are un model. O mulțime Γ de formule este **satisfiabilă** dacă are un model.
- O formulă φ este **tautologie** (**validă, universal adevarată**) dacă $e^+(\varphi) = 1$ pentru orice evaluare $e : Var \rightarrow \{0, 1\}$.
Notăm prin $\models \varphi$ faptul că φ este o tautologie.
- O formulă φ este **Γ -tautologie** (**consecință semantică a lui Γ**) dacă orice model al lui Γ este și model pentru φ , i.e. $e^+(\Gamma) = \{1\}$ implică $e^+(\varphi) = 1$ pentru orice evaluare $e : Var \rightarrow \{0, 1\}$.
Notăm prin $\Gamma \models \varphi$ faptul că φ este o Γ -tautologie.

Semantica PL

Cum verificăm că o formulă este tautologie: $\models \varphi$?

- Fie v_1, \dots, v_n variabilele care apar în φ .
- Cele 2^n evaluări posibile e_1, \dots, e_{2^n} pot fi scrise într-un tabel:

Semantica PL

Cum verificăm că o formulă este tautologie: $\models \varphi$?

- Fie v_1, \dots, v_n variabilele care apar în φ .
- Cele 2^n evaluări posibile e_1, \dots, e_{2^n} pot fi scrise într-un tabel:

v_1	v_2	\dots	v_n	φ
$e_1(v_1)$	$e_1(v_2)$	\dots	$e_1(v_n)$	$e_1^+(\varphi)$
$e_2(v_1)$	$e_2(v_2)$	\dots	$e_2(v_n)$	$e_2^+(\varphi)$
\vdots	\vdots	\vdots	\vdots	\vdots
$e_{2^n}(v_1)$	$e_{2^n}(v_2)$	\dots	$e_{2^n}(v_n)$	$e_{2^n}^+(\varphi)$

Fiecare evaluare corespunde unei linii din tabel!

Semantica PL

Cum verificăm că o formulă este tautologie: $\models \varphi$?

- Fie v_1, \dots, v_n variabilele care apar în φ .
- Cele 2^n evaluări posibile e_1, \dots, e_{2^n} pot fi scrise într-un tabel:

v_1	v_2	\dots	v_n	φ
$e_1(v_1)$	$e_1(v_2)$	\dots	$e_1(v_n)$	$e_1^+(\varphi)$
$e_2(v_1)$	$e_2(v_2)$	\dots	$e_2(v_n)$	$e_2^+(\varphi)$
\vdots	\vdots	\vdots	\vdots	\vdots
$e_{2^n}(v_1)$	$e_{2^n}(v_2)$	\dots	$e_{2^n}(v_n)$	$e_{2^n}^+(\varphi)$

Fiecare evaluare corespunde unei linii din tabel!

- $\models \varphi$ dacă și numai dacă $e_1^+(\varphi) = \dots = e_{2^n}^+(\varphi) = 1$

Verificarea problemei consecinței logice

- În principiu, putem verifica problema consecinței logice construind un **tabel de adevăr**, cu câte o linie pentru fiecare interpretare posibilă.

Verificarea problemei consecinței logice

- În principiu, putem verifica problema consecinței logice construind un **tabel de adevăr**, cu câte o linie pentru fiecare interpretare posibilă.
- În cazul în care formula conțin n variabile, tabelul de adevăr are 2^n rânduri. Această metodă este atât de costisitoare computațional, încât este irealizabilă. (**Timp exponential**)

Verificarea problemei consecinței logice

- În principiu, putem verifica problema consecinței logice construind un **tabel de adevăr**, cu câte o linie pentru fiecare interpretare posibilă.
- În cazul în care formula conțin n variabile, tabelul de adevăr are 2^n rânduri. Această metodă este atât de costisitoare computațional, încât este irealizabilă. (**Timp exponential**)
- **Problemă deschisă de un milion de dolari:**
Este posibil să decidem problema consecinței logice în cazul propozițional printr-un algoritm care să funcționeze în timp polinomial?

Verificarea problemei consecinței logice

- În principiu, putem verifica problema consecinței logice construind un **tabel de adevăr**, cu câte o linie pentru fiecare interpretare posibilă.
- În cazul în care formula conțin n variabile, tabelul de adevăr are 2^n rânduri. Această metodă este atât de costisitoare computațional, încât este irealizabilă. (**Timp exponential**)

- Problemă deschisă de un milion de dolari:

Este posibil să decidem problema consecinței logice în cazul propozițional printr-un algoritm care să funcționeze în timp polinomial?

- Echivalent, este adevărată $P = NP$?

(Institutul de Matematica Clay – Millennium Prize Problems)

Verificarea problemei consecinței logice

- În principiu, putem verifica problema consecinței logice construind un **tabel de adevăr**, cu câte o linie pentru fiecare interpretare posibilă.
- În cazul în care formula conțin n variabile, tabelul de adevăr are 2^n rânduri. Această metodă este atât de costisitoare computațional, încât este irealizabilă. (**Timp exponential**)
- **Problemă deschisă de un milion de dolari:**

Este posibil să decidem problema consecinței logice în cazul propozițional printr-un algoritm care să funcționeze în timp polinomial?

- **Echivalent**, este adevărată $P = NP$?
(Institutul de Matematica Clay – Millennium Prize Problems)
- **SAT** este problema satisfiabilității în calculul propozițional clasic.
SAT-solverele sunt bazate pe metode sintactice.

Sintaxa PL

Sisteme deductive pentru calculul propozițional clasic:

- Sistemul Hilbert
- Rezoluție
- Deducreția naturală
- Sistemul Gentzen

Sistemul Hilbert

- Oricare ar fi $\varphi, \psi, \chi \in Form$ următoarele formule sunt **axiome**:
 - (A1) $\varphi \rightarrow (\psi \rightarrow \varphi)$
 - (A2) $(\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))$
 - (A3) $(\neg\psi \rightarrow \neg\varphi) \rightarrow (\varphi \rightarrow \psi).$
- Regula de deducție este **modus ponens**:
$$\frac{\varphi, \varphi \rightarrow \psi}{\psi} \text{MP}$$

Sistemul Hilbert

- Oricare ar fi $\varphi, \psi, \chi \in Form$ următoarele formule sunt **axiome**:

$$(A1) \quad \varphi \rightarrow (\psi \rightarrow \varphi)$$

$$(A2) \quad (\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))$$

$$(A3) \quad (\neg\psi \rightarrow \neg\varphi) \rightarrow (\varphi \rightarrow \psi).$$

- **Regula de deducție** este **modus ponens**:
$$\frac{\varphi, \varphi \rightarrow \psi}{\psi} MP$$

- O **demonstrație** pentru φ este o secvență de formule $\gamma_1, \dots, \gamma_n$ astfel încât $\gamma_n = \varphi$ și, pentru fiecare $i \in \{1, \dots, n\}$, una din următoarele condiții este satisfăcută:

- γ_i este axiomă,
- γ_i se obține din formulele anterioare prin **MP**:
există $j, k < i$ astfel încât $\gamma_j = \gamma_k \rightarrow \gamma_i$

Sistemul Hilbert

- Oricare ar fi $\varphi, \psi, \chi \in Form$ următoarele formule sunt **axiome**:

$$(A1) \quad \varphi \rightarrow (\psi \rightarrow \varphi)$$

$$(A2) \quad (\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))$$

$$(A3) \quad (\neg\psi \rightarrow \neg\varphi) \rightarrow (\varphi \rightarrow \psi).$$

- **Regula de deducție** este **modus ponens**:
$$\frac{\varphi, \varphi \rightarrow \psi}{\psi} MP$$

- O **demonstrație** pentru φ este o secvență de formule $\gamma_1, \dots, \gamma_n$ astfel încât $\gamma_n = \varphi$ și, pentru fiecare $i \in \{1, \dots, n\}$, una din următoarele condiții este satisfăcută:

- γ_i este axiomă,
- γ_i se obține din formulele anterioare prin **MP**:
există $j, k < i$ astfel încât $\gamma_j = \gamma_k \rightarrow \gamma_i$

- O formulă φ este **teoremă** dacă are o demonstrație.
Notăm prin $\vdash \varphi$ faptul că φ este teoremă.

Sistemul Hilbert

(A1) $\varphi \rightarrow (\psi \rightarrow \varphi)$

(A2) $(\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))$

(A3) $(\neg\psi \rightarrow \neg\varphi) \rightarrow (\varphi \rightarrow \psi).$

Regula de deducție este modus ponens: $\frac{\varphi, \varphi \rightarrow \psi}{\psi}$ MP

Example

Fie φ și ψ formule în logica propozițională. Să se arate sintactic că

$$\vdash \varphi \rightarrow \varphi.$$

Sistemul Hilbert

(A1) $\varphi \rightarrow (\psi \rightarrow \varphi)$

(A2) $(\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))$

(A3) $(\neg\psi \rightarrow \neg\varphi) \rightarrow (\varphi \rightarrow \psi).$

Regula de deducție este modus ponens: $\frac{\varphi, \varphi \rightarrow \psi}{\psi}$ MP

Example

Fie φ și ψ formule în logica propozițională. Să se arate sintactic că

$$\vdash \varphi \rightarrow \varphi.$$

Avem următoarea demonstrație:

$$(1) \quad \varphi \rightarrow ((\varphi \rightarrow \varphi) \rightarrow \varphi) \quad (\text{A1})$$

$$(2) \quad (\varphi \rightarrow ((\varphi \rightarrow \varphi) \rightarrow \varphi)) \rightarrow ((\varphi \rightarrow (\varphi \rightarrow \varphi)) \rightarrow (\varphi \rightarrow \varphi)) \quad (\text{A2})$$

$$(3) \quad (\varphi \rightarrow (\varphi \rightarrow \varphi)) \rightarrow (\varphi \rightarrow \varphi) \quad (\text{MP})$$

$$(4) \quad (\varphi \rightarrow (\varphi \rightarrow \varphi)) \quad (\text{A1})$$

$$(5) \quad (\varphi \rightarrow \varphi) \quad (\text{MP})$$

Sistemul Hilbert

- Oricare ar fi $\varphi, \psi, \chi \in Form$ următoarele formule sunt **axiome**:
 - (A1) $\varphi \rightarrow (\psi \rightarrow \varphi)$
 - (A2) $(\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))$
 - (A3) $(\neg\psi \rightarrow \neg\varphi) \rightarrow (\varphi \rightarrow \psi)$.
- Regula de deducție este **modus ponens**:
$$\frac{\varphi, \varphi \rightarrow \psi}{\psi} \text{MP}$$

Teorema de completitudine

Teoremele și tautologiile coincid, i.e. pentru orice $\varphi \in Form$ avem

$$\vdash \varphi \text{ dacă și numai dacă } \vDash \varphi$$

(\Rightarrow) **Corectitudine**

(\Leftarrow) **Completitudine**

PL - Deductie naturala

Deductia naturală¹ pe scurt

- În deductia naturală deducem (demonstrăm) formule din alte formule folosind reguli de deductie.

¹M. Huth, M. Ryan, Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge University Press New York, 2004.

Deductia naturală¹ pe scurt

- În deductia naturală deducem (demonstrăm) formule din alte formule folosind reguli de deductie.
- Numim secent o expresie de forma

$$\varphi_1, \dots, \varphi_n \vdash \psi$$

Formulele $\varphi_1, \dots, \varphi_n$ se numesc premise, iar ψ se numește concluzie.

¹M. Huth, M. Ryan, Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge University Press New York, 2004.

Deductia naturală¹ pe scurt

- În deductia naturală deducem (demonstrăm) formule din alte formule folosind reguli de deducție.
- Numim **secvent** o expresie de forma

$$\varphi_1, \dots, \varphi_n \vdash \psi$$

Formulele $\varphi_1, \dots, \varphi_n$ se numesc **premise**, iar ψ se numește **concluzie**.

- Un secvent este **valid** dacă există o demonstrație folosind regulile de deducție.

¹M. Huth, M. Ryan, Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge University Press New York, 2004.

Deductia naturală¹ pe scurt

- În deductia naturală deducem (demonstrăm) formule din alte formule folosind reguli de deducție.
- Numim **secvent** o expresie de forma

$$\varphi_1, \dots, \varphi_n \vdash \psi$$

Formulele $\varphi_1, \dots, \varphi_n$ se numesc **premise**, iar ψ se numește **concluzie**.

- Un secvent este **valid** dacă există o demonstrație folosind regulile de deducție.
- O **teoremă** este o formulă ψ astfel încât $\vdash \psi$
(adică ψ poate fi demonstrată din mulțimea vidă de ipoteze).

¹M. Huth, M. Ryan, Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge University Press New York, 2004.

Deductia naturală¹ pe scurt

- În deductia naturală deducem (demonstrăm) formule din alte formule folosind reguli de deducție.
- Numim **secvent** o expresie de forma

$$\varphi_1, \dots, \varphi_n \vdash \psi$$

Formulele $\varphi_1, \dots, \varphi_n$ se numesc **premise**, iar ψ se numește **concluzie**.

- Un secvent este **valid** dacă există o demonstrație folosind regulile de deducție.
- O **teoremă** este o formulă ψ astfel încât $\vdash \psi$
(adică ψ poate fi demonstrată din mulțimea vidă de ipoteze).
- Pentru fiecare conector logic vom avea reguli de introducere și reguli de eliminare.

¹M. Huth, M. Ryan, Logic in Computer Science: Modelling and Reasoning about Systems, Cambridge University Press New York, 2004.

Regulile pentru conjuncție

- Intuitiv, a demonstra $\varphi \wedge \psi$ revine la a demonstra φ și ψ . Obținem astfel regula

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi} \text{ (\wedge i)}$$

Eticheta $(\wedge i)$ înseamnă \wedge -introducere deoarece \wedge este introdus în concluzie.

Regulile pentru conjuncție

- Intuitiv, a demonstra $\varphi \wedge \psi$ revine la a demonstra φ și ψ . Obținem astfel regula

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi} \text{ (\wedge i)}$$

Eticheta $(\wedge i)$ înseamnă \wedge -introducere deoarece \wedge este introdus în concluzie.

- Regulile pentru \wedge -eliminare sunt:

$$\frac{\varphi \wedge \psi}{\varphi} \text{ (\wedge e}_1\text{)} \quad \frac{\varphi \wedge \psi}{\psi} \text{ (\wedge e}_2\text{)}$$

Regulile pentru conjuncție

Example

Demonstrați că secventul $p \wedge q, r \vdash q \wedge r$ este valid.

Regulile pentru conjuncție

Example

Demonstrați că secventul $p \wedge q, r \vdash q \wedge r$ este valid.

Putem scrie demonstrația ca un [arbore](#)

$$\frac{\frac{p \wedge q}{q} \text{ } (\wedge e_2) \quad r}{q \wedge r} \text{ } (\wedge i)$$

Regulile pentru conjuncție

Example

Demonstrați că secventul $p \wedge q, r \vdash q \wedge r$ este valid.

Putem scrie demonstrația ca un [arbore](#)

$$\frac{\frac{p \wedge q}{q} \text{ } (\wedge e_2) \quad r}{q \wedge r} \text{ } (\wedge i)$$

sau putem scrie demonstrația într-un [mod liniar](#) astfel:

1	$p \wedge q$	premisa
2	r	premisa

Regulile pentru conjuncție

Example

Demonstrați că secventul $p \wedge q, r \vdash q \wedge r$ este valid.

Putem scrie demonstrația ca un [arbore](#)

$$\frac{\frac{p \wedge q}{q} \quad (\wedge e_2) \quad r}{q \wedge r} \quad (\wedge i)$$

sau putem scrie demonstrația într-un [mod liniar](#) astfel:

1	$p \wedge q$	premisa
2	r	premisa
3	q	$(\wedge e_2), 1$

Regulile pentru conjuncție

Example

Demonstrați că secventul $p \wedge q, r \vdash q \wedge r$ este valid.

Putem scrie demonstrația ca un [arbore](#)

$$\frac{\frac{p \wedge q}{q} \quad (\wedge e_2) \quad r}{q \wedge r} \quad (\wedge i)$$

sau putem scrie demonstrația într-un [mod liniar](#) astfel:

1	$p \wedge q$	premisa
2	r	premisa
3	q	$(\wedge e_2), 1$
4	$q \wedge r$	$(\wedge i), 3, 2$

Regulile pentru dubla negație

- Regulile $\neg\neg$ -introducere și $\neg\neg$ -eliminare sunt:

$$\frac{\neg\neg\varphi}{\varphi} \text{ (\neg\neg e)} \quad \frac{\varphi}{\neg\neg\varphi} \text{ (\neg\neg i)}$$

Regulile pentru dubla negație

- Regulile $\neg\neg$ -introducere și $\neg\neg$ -eliminare sunt:

$$\frac{\neg\neg\varphi}{\varphi} \text{ (\neg\neg e)} \quad \frac{\varphi}{\neg\neg\varphi} \text{ (\neg\neg i)}$$

Example

Demonstrați că secientul $\neg\neg(q \wedge r) \vdash \neg\neg r$ este valid.

Regulile pentru dubla negație

- Regulile $\neg\neg$ -introducere și $\neg\neg$ -eliminare sunt:

$$\frac{\neg\neg\varphi}{\varphi} \text{ (\neg\neg e)} \quad \frac{\varphi}{\neg\neg\varphi} \text{ (\neg\neg i)}$$

Example

Demonstrați că seceventul $\neg\neg(q \wedge r) \vdash \neg\neg r$ este valid.

1	$\neg\neg(q \wedge r)$	premisa
2	$q \wedge r$	$(\neg\neg ei), 1$
3	r	$(\wedge ei_2), 2$

Regulile pentru dubla negație

- Regulile $\neg\neg$ -introducere și $\neg\neg$ -eliminare sunt:

$$\frac{\neg\neg\varphi}{\varphi} \text{ (\neg\neg e)} \quad \frac{\varphi}{\neg\neg\varphi} \text{ (\neg\neg i)}$$

Example

Demonstrați că seceventul $\neg\neg(q \wedge r) \vdash \neg\neg r$ este valid.

1	$\neg\neg(q \wedge r)$	premisa
2	$q \wedge r$	($\neg\neg ei$),1
3	r	($\wedge ei_2$),2
4	$\neg\neg r$	($\neg\neg i$),3

Regulile pentru implicație: \rightarrow -eliminare

- Regula de \rightarrow -eliminare o stii deja:

Regulile pentru implicație: \rightarrow -eliminare

- Regula de \rightarrow -eliminare o stii deja: este modus ponens:

$$\frac{\varphi \quad \varphi \rightarrow \psi}{\psi} (\rightarrow e)$$

Regulile pentru implicație: \rightarrow -introducere

- Intuitiv, a demonstra $\varphi \rightarrow \psi$ revine la a demonstra ψ în ipoteza φ , i.e. presupunem temporar φ și demonstrăm ψ .

Regulile pentru implicație: \rightarrow -introducere

- Intuitiv, a demonstra $\varphi \rightarrow \psi$ revine la a demonstra ψ în ipoteza φ , i.e. presupunem temporar φ și demonstrăm ψ .

Acum lucru se reprezintă astfel:

$$\frac{\varphi \quad \vdots \quad \psi}{\varphi \rightarrow \psi} (\rightarrow i)$$

Regulile pentru implicație: \rightarrow -introducere

- Intuitiv, a demonstra $\varphi \rightarrow \psi$ revine la a demonstra ψ în ipoteza φ , i.e. presupunem temporar φ și demonstrăm ψ .
Acum lucru se reprezintă astfel:

$$\frac{\varphi \quad \vdots \quad \psi}{\varphi \rightarrow \psi} (\rightarrow i)$$

- Cutia (chenarul) are rostul de a marca scopul ipotezei φ : numai deducțiile din interiorul cutiei pot folosi φ .
- În momentul în care am obținut ψ , închidem cutia și deducem $\varphi \rightarrow \psi$ în afara cutiei.
- O ipoteză nu poate fi folosită în afara scopului său.

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \wedge q) \rightarrow p$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \wedge q) \rightarrow p$

Vom considera $p \wedge q$ ca ipoteză temporară

$$\boxed{p \wedge q}$$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \wedge q) \rightarrow p$

Vom considera $p \wedge q$ ca ipoteză temporară

$$\boxed{\frac{p \wedge q}{p} (\wedge e_1)}$$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \wedge q) \rightarrow p$

Vom considera $p \wedge q$ ca ipoteză temporară

$$\boxed{\frac{p \wedge q}{p} (\wedge e_1)}$$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \wedge q) \rightarrow p$

Vom considera $p \wedge q$ ca ipoteză temporară

$$\frac{\frac{p \wedge q}{p} (\wedge e_1)}{p \wedge q \rightarrow p} (\rightarrow i)$$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \wedge q) \rightarrow p$

Putem scrie demonstrația într-un mod liniar în felul următor:

$$1 \quad \overline{p \wedge q} \qquad \text{ipoteza}$$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \wedge q) \rightarrow p$

Putem scrie demonstrația într-un mod liniar în felul următor:

1	$p \wedge q$	ipoteza
2	p	$(\wedge e_1), 1$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \wedge q) \rightarrow p$

Putem scrie demonstrația într-un mod liniar în felul următor:

1	$p \wedge q$	ipoteza
2	p	$(\wedge e_1), 1$
3	$p \wedge q \rightarrow p$	$(\rightarrow i), 1-2$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash p \rightarrow p$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash p \rightarrow p$

1	p	ipoteza
2	$p \rightarrow p$	$(\rightarrow i), 1$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \rightarrow q) \rightarrow ((q \rightarrow r) \rightarrow (p \rightarrow r))$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \rightarrow q) \rightarrow ((q \rightarrow r) \rightarrow (p \rightarrow r))$

1	$p \rightarrow q$	ipoteza
2	$q \rightarrow r$	ipoteza
3	p	ipoteza

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \rightarrow q) \rightarrow ((q \rightarrow r) \rightarrow (p \rightarrow r))$

1	$p \rightarrow q$	ipoteza
2	$q \rightarrow r$	ipoteza
3	p	ipoteza
4	q	$(\rightarrow e), 1, 3$
5	r	$(\rightarrow e), 2, 4$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \rightarrow q) \rightarrow ((q \rightarrow r) \rightarrow (p \rightarrow r))$

1	$p \rightarrow q$	ipoteza
2	$q \rightarrow r$	ipoteza
3	p	ipoteza
4	q	$(\rightarrow e), 1, 3$
5	r	$(\rightarrow e), 2, 4$
6	$p \rightarrow r$	$(\rightarrow i), 3-5$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \rightarrow q) \rightarrow ((q \rightarrow r) \rightarrow (p \rightarrow r))$

1	$p \rightarrow q$	ipoteza
2	$q \rightarrow r$	ipoteza
3	p	ipoteza
4	q	$(\rightarrow e), 1, 3$
5	r	$(\rightarrow e), 2, 4$
6	$p \rightarrow r$	$(\rightarrow i), 3-5$
7	$(q \rightarrow r) \rightarrow (p \rightarrow r)$	$(\rightarrow i), 2-6$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \rightarrow q) \rightarrow ((q \rightarrow r) \rightarrow (p \rightarrow r))$

1	$p \rightarrow q$	ipoteza
2	$q \rightarrow r$	ipoteza
3	p	ipoteza
4	q	$(\rightarrow e), 1, 3$
5	r	$(\rightarrow e), 2, 4$
6	$p \rightarrow r$	$(\rightarrow i), 3-5$
7	$(q \rightarrow r) \rightarrow (p \rightarrow r)$	$(\rightarrow i), 2-6$
8	$(p \rightarrow q) \rightarrow ((q \rightarrow r) \rightarrow (p \rightarrow r))$	$(\rightarrow i), 1-7$

Regulile "cutiilor"

- O cutie marchează scopul unei ipoteze temporare, ce poate fi folosită pentru a demonstra formulele din interiorul cutiei.

Regulile "cutiilor"

- O cutie marchează scopul unei ipoteze temporare, ce poate fi folosită pentru a demonstra formulele din interiorul cutiei.
- Cutiile pot fi incluse una în alta; se pot deschide cutii noi după închiderea celor vechi.

Regulile "cutiilor"

- O cutie marchează scopul unei ipoteze temporare, ce poate fi folosită pentru a demonstra formulele din interiorul cutiei.
- Cutiile pot fi incluse una în alta; se pot deschide cutii noi după închiderea celor vechi.
- Linia care urmează după închiderea unei cutii trebuie să conțină concluzia regulii pentru care a fost utilizată cutia.

Regulile "cutiilor"

- O cutie marchează scopul unei ipoteze temporare, ce poate fi folosită pentru a demonstra formulele din interiorul cutiei.
- Cutiile pot fi incluse una în alta; se pot deschide cutii noi după închiderea celor vechi.
- Linia care urmează după închiderea unei cutii trebuie să conțină concluzia regulii pentru care a fost utilizată cutia.
- Într-un punct al unei demonstrații se pot folosi formulele care au apărut anterior, cu excepția celor din interiorul cutiilor închise.

Regula pentru "copiere"

- La un pas al unei demonstrații poate fi copiată orice formulă demonstrată anterior.
- La un pas al unei demonstrații nu pot fi copiate formule din interiorul cutiilor care sunt închise în acel moment.

Regula pentru "copiere"

- La un pas al unei demonstrații poate fi copiată orice formulă demonstrată anterior.
- La un pas al unei demonstrații nu pot fi copiate formule din interiorul cutiilor care sunt închise în acel moment.

Example

Demonstrați teorema $\vdash p \rightarrow (q \rightarrow p)$

Regula pentru "copiere"

- La un pas al unei demonstrații poate fi copiată orice formulă demonstrată anterior.
- La un pas al unei demonstrații nu pot fi copiate formule din interiorul cutiilor care sunt închise în acel moment.

Example

Demonstrați teorema $\vdash p \rightarrow (q \rightarrow p)$

1	p	ipoteza
2	q	ipoteza

Regula pentru "copiere"

- La un pas al unei demonstrații poate fi copiată orice formulă demonstrată anterior.
- La un pas al unei demonstrații nu pot fi copiate formule din interiorul cutiilor care sunt închise în acel moment.

Example

Demonstrați teorema $\vdash p \rightarrow (q \rightarrow p)$

1	p	ipoteza
2	q	ipoteza
3	p	copiere 1

Regula pentru "copiere"

- La un pas al unei demonstrații poate fi copiată orice formulă demonstrată anterior.
- La un pas al unei demonstrații nu pot fi copiate formule din interiorul cutiilor care sunt închise în acel moment.

Example

Demonstrați teorema $\vdash p \rightarrow (q \rightarrow p)$

1	p	<i>ipoteza</i>
2	q	<i>ipoteza</i>
3	p	<i>copiere 1</i>
4	$q \rightarrow p$	$(\rightarrow i), 2-3$

Regula pentru "copiere"

- La un pas al unei demonstrații poate fi copiată orice formulă demonstrată anterior.
- La un pas al unei demonstrații nu pot fi copiate formule din interiorul cutiilor care sunt închise în acel moment.

Example

Demonstrați teorema $\vdash p \rightarrow (q \rightarrow p)$

1	p	ipoteza
2	q	ipoteza
3	p	copiere 1
4	$q \rightarrow p$	$(\rightarrow i), 2-3$
5	$p \rightarrow (q \rightarrow p)$	$(\rightarrow i), 1-4$

Regulile pentru disjuncție: \vee -introducere

- Intuitiv, a demonstra $\varphi \vee \psi$ revine la a demonstra φ sau ψ . În consecință, regulile de \vee -introducere sunt

$$\frac{\varphi}{\varphi \vee \psi} \text{ (}\vee i_1\text{)} \quad \frac{\psi}{\varphi \vee \psi} \text{ (}\vee i_2\text{)}$$

Regulile pentru disjuncție: \vee -introducere

- Intuitiv, a demonstra $\varphi \vee \psi$ revine la a demonstra φ sau ψ . În consecință, regulile de \vee -introducere sunt

$$\frac{\varphi}{\varphi \vee \psi} \text{ (}\vee i_1\text{)} \quad \frac{\psi}{\varphi \vee \psi} \text{ (}\vee i_2\text{)}$$

Example

Demonstrați că secientul $q \rightarrow r \vdash q \rightarrow (r \vee p)$ este valid.

Regulile pentru disjuncție: \vee -introducere

- Intuitiv, a demonstra $\varphi \vee \psi$ revine la a demonstra φ sau ψ . În consecință, regulile de **\vee -introducere** sunt

$$\frac{\varphi}{\varphi \vee \psi} \text{ (}\vee i_1\text{)} \quad \frac{\psi}{\varphi \vee \psi} \text{ (}\vee i_2\text{)}$$

Example

Demonstrați că seceventul $q \rightarrow r \vdash q \rightarrow (r \vee p)$ este valid.

1	$q \rightarrow r$	premisa
2	q	ipoteza
3	r	$(\rightarrow e), 1, 2$
4	$r \vee p$	$(\vee i_1), 3$
5	$q \rightarrow (r \vee p)$	$(\rightarrow i), 2-4$

Regulile pentru disjuncție: \vee -eliminare

- Cum procedăm pentru a demonstra χ știind $\varphi \vee \psi$?

Trebuie să analizăm două cazuri:

- presupunem φ și demonstrăm χ
 - presupunem ψ și demonstrăm χ

Astfel, dacă am demonstrat $\varphi \vee \psi$ putem să deducem χ deoarece cazurile de mai sus acoperă toate situațiile posibile.

Regulile pentru disjuncție: \vee -eliminare

- Cum procedăm pentru a demonstra χ știind $\varphi \vee \psi$?

Trebuie să analizăm două cazuri:

- presupunem φ și demonstrăm χ
- presupunem ψ și demonstrăm χ

Astfel, dacă am demonstrat $\varphi \vee \psi$ putem să deducem χ deoarece cazurile de mai sus acoperă toate situațiile posibile.

- Regula \vee -eliminare reflectă această argumentație:

$$\frac{\begin{array}{c} \varphi \\ \vdots \\ \varphi \vee \psi \end{array} \quad \begin{array}{c} \psi \\ \vdots \\ \varphi \vee \psi \end{array}}{\chi} (\vee e)$$

Regulile pentru disjuncție

Example

Demonstrați că secventul $q \rightarrow r \vdash (p \vee q) \rightarrow (p \vee r)$ este valid.

1	$q \rightarrow r$	premisa
2	$p \vee q$	ipoteza
3	p	ipoteza
4	$p \vee r$	$(\vee i_1), 3$
5	q	ipoteza
6	r	$(\rightarrow e), 1, 5$
7	$p \vee r$	$(\vee i_2), 6$
8	$p \vee r$	$(\vee e), 2, 3-4, 5-7$
9	$p \vee q \rightarrow p \vee r$	$(\rightarrow i), 2-8$

Regulile pentru negație

- Pentru orice φ , formulele $\varphi \wedge \neg\varphi$ și $\neg\varphi \wedge \varphi$ se numesc **contradicții**. O contradicție arbitrară va fi notată \perp .

Regulile pentru negație

- Pentru orice φ , formulele $\varphi \wedge \neg\varphi$ și $\neg\varphi \wedge \varphi$ se numesc **contradicții**. O contradicție arbitrară va fi notată \perp .
- Faptul că dintr-o contradicție se poate deduce orice este reprezentat printr-o regulă specială:

$$\frac{\perp}{\varphi} (\perp e)$$

Regulile pentru negație

- Pentru orice φ , formulele $\varphi \wedge \neg\varphi$ și $\neg\varphi \wedge \varphi$ se numesc **contradicții**. O contradicție arbitrară va fi notată \perp .
- Faptul că dintr-o contradicție se poate deduce orice este reprezentat printr-o regulă specială:

$$\frac{\perp}{\varphi} (\perp e)$$

- Regulile de **\neg -eliminare** și **\neg -introducere** sunt:

$$\frac{\varphi \quad \neg\varphi}{\perp} (\neg e)$$

$$\frac{\boxed{\begin{array}{c}\varphi \\ \vdots \\ \perp\end{array}}}{\neg\varphi} (\neg i)$$

Regulile pentru negație

Example

Demonstrați că seceventul $p \rightarrow \neg p \vdash \neg p$ este valid.

1	$p \rightarrow \neg p$	premisa
2	p	ipoteza
3	$\neg p$	$(\rightarrow e), 1, 2$
4	\perp	$(\neg e), 2, 3$
5	$\neg p$	$(\neg i), 2-4$

Regulile DN

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi} \text{ (\wedge i)}$$

$$\frac{\begin{array}{c} \varphi \\ \vdots \\ \psi \end{array}}{\varphi \rightarrow \psi} \text{ (\rightarrow i)}$$

$$\frac{\varphi}{\varphi \vee \psi} \text{ (\vee i}_1)$$

$$\frac{\psi}{\varphi \vee \psi} \text{ (\vee i}_2)$$

$$\frac{\varphi}{\neg \neg \varphi} \text{ (\neg \neg i)}$$

$$\frac{\begin{array}{c} \varphi \\ \vdots \\ \perp \end{array}}{\neg \varphi} \text{ (\neg i)}$$

$$\frac{\varphi \wedge \psi}{\varphi} \text{ (\wedge e}_1)$$

$$\frac{\varphi \wedge \psi}{\psi} \text{ (\wedge e}_2)$$

$$\frac{\varphi \quad \varphi \rightarrow \psi}{\psi} \text{ (\rightarrow e)}$$

$$\frac{\varphi \vee \psi}{\begin{array}{c|c} \varphi & \psi \\ \vdots & \vdots \\ \chi & \chi \end{array}} \text{ (\vee e)}$$

$$\frac{\neg \neg \varphi}{\varphi} \text{ (\neg \neg e)}$$

$$\frac{\varphi \quad \neg \varphi}{\perp} \text{ (\neg e)}$$

$$\frac{\perp}{\varphi} \text{ (\perp e)}$$

Reguli derivate

- Următoarele reguli pot fi derivate din regulile deductiei naturale:

$$\frac{\varphi \rightarrow \psi \quad \neg\psi}{\neg\varphi} \text{ MT}$$

$$\frac{\neg\varphi \quad \vdots \quad \perp}{\varphi} \text{ RAA}$$

$$\frac{}{\varphi \vee \neg\varphi} \text{ TND}$$

Deductia naturală DN

- este un sistem deductiv corect și complet pentru logica clasică,
- stabilește reguli de deducție pentru fiecare operator logic,
- o demonstrație se construiește prin aplicarea succesivă a regulilor de deducție,
- în demonstrații putem folosi ipoteze temporare, scopul acestora fiind bine delimitat.

PL - Deducție naturală: Corectitudinea

Corectitudinea DN

Teoremă

Deduclia naturală este corectă, i.e.

dacă $\varphi_1, \dots, \varphi_n \vdash \varphi$ este valid atunci $\varphi_1, \dots, \varphi_n \vDash \varphi$

oricare ar fi $n \geq 0$ și formulele $\varphi_1, \dots, \varphi_n, \varphi$.

Corectitudinea DN

Teoremă

Deduclia naturală este corectă, i.e.

dacă $\varphi_1, \dots, \varphi_n \vdash \varphi$ este valid atunci $\varphi_1, \dots, \varphi_n \vDash \varphi$

oricare ar fi $n \geq 0$ și formulele $\varphi_1, \dots, \varphi_n, \varphi$.

Demonstrație

Din ipoteză știm că există o demonstrație pentru φ din ipotezele $\varphi_1, \dots, \varphi_n$ folosind regulile deducției naturale.

Corectitudinea DN

Teoremă

Deduclia naturală este corectă, i.e.

dacă $\varphi_1, \dots, \varphi_n \vdash \varphi$ este valid atunci $\varphi_1, \dots, \varphi_n \vDash \varphi$

oricare ar fi $n \geq 0$ și formulele $\varphi_1, \dots, \varphi_n, \varphi$.

Demonstrație

Din ipoteză știm că există o demonstrație pentru φ din ipotezele $\varphi_1, \dots, \varphi_n$ folosind regulile deducției naturale.

Fie k numărul de linii dintr-o demonstrație în forma liniară.

Corectitudinea DN

Teoremă

Deduclia naturală este corectă, i.e.

dacă $\varphi_1, \dots, \varphi_n \vdash \varphi$ este valid atunci $\varphi_1, \dots, \varphi_n \vDash \varphi$

oricare ar fi $n \geq 0$ și formulele $\varphi_1, \dots, \varphi_n, \varphi$.

Demonstrație

Din ipoteză știm că există o demonstrație pentru φ din ipotezele $\varphi_1, \dots, \varphi_n$ folosind regulile deducției naturale.

Fie k numărul de linii dintr-o demonstrație în forma liniară. Prin inducție după $k \geq 1$ vom arăta că

oricare ar fi $n \geq 0$ și $\varphi_1, \dots, \varphi_n, \varphi$ formule, dacă $\varphi_1, \dots, \varphi_n \vdash \varphi$ are o demonstrație de lungime $k \geq 1$ atunci $\varphi_1, \dots, \varphi_n \vDash \varphi$,

(orice secvent care are o demonstrație de lungime k este corect).

Corectitudinea DN

Demonstrație (cont.)

Atenție! Facem inducție după lungimea demonstrației, numărul de premise este arbitrar.

Corectitudinea DN

Demonstrație (cont.)

Atenție! Facem inducție după lungimea demonstrației, numărul de premise este arbitrar. Cazul $k = 1$. În acest caz demonstrația este

1 φ premisa

ceea ce înseamnă că secientul inițial este $\varphi \vdash \varphi$.

Este evident că $\varphi \models \varphi$

Corectitudinea DN

Demonstrație (cont.)

Cazul de inducție. Vom presupune că:

oricare ar fi $\varphi_1, \dots, \varphi_n, \varphi$, dacă $\varphi_1, \dots, \varphi_n \vdash \varphi$ are o demonstrație de lungime $< k$ atunci $\varphi_1, \dots, \varphi_n \vdash \varphi$

și vom demonstra că proprietatea este adevărată pentru secvenții cu demonstrații de lungime k .

Corectitudinea DN

Demonstrație (cont.)

Cazul de inducție. Vom presupune că:

oricare ar fi $\varphi_1, \dots, \varphi_n, \varphi$, dacă $\varphi_1, \dots, \varphi_n \vdash \varphi$ are o demonstrație de lungime $< k$ atunci $\varphi_1, \dots, \varphi_n \vdash \varphi$

și vom demonstra că proprietatea este adevărată pentru secvenții cu demonstrații de lungime k .

Fie (R) ultima regulă care se aplică în demonstrație, adică

1	φ_1	<i>premisa</i>
	:	
n	φ_n	<i>premisa</i>
	:	
k	φ	(R)

Corectitudinea DN

Demonstrație (cont.)

Presupunem că ultima regulă a fost $(\wedge i)$. Aceasta înseamnă că

$$\varphi = \psi \wedge \chi$$

Corectitudinea DN

Demonstrație (cont.)

Presupunem că ultima regulă a fost $(\wedge i)$. Aceasta înseamnă că

$$\varphi = \psi \wedge \chi$$

1 φ_1 *premisa*

⋮

n φ_n *premisa*

⋮

k_1 ψ

⋮

k_2 χ

k $\psi \wedge \chi$ $(\wedge i)k_1, k_2$

Corectitudinea DN

Demonstrație (cont.)

Presupunem că ultima regulă a fost $(\wedge i)$. Aceasta înseamnă că

$$\varphi = \psi \wedge \chi$$

1 φ_1 *premisa*

Se observă că secvenții

⋮

$\varphi_1, \dots, \varphi_n \vdash \psi$ și

n φ_n *premisa*

$\varphi_1, \dots, \varphi_n \vdash \chi$

⋮

au demonstrații de lungime $< k$.

k_1 ψ

⋮

k_2 χ

k $\psi \wedge \chi$ $(\wedge i)k_1, k_2$

Corectitudinea DN

Demonstrație (cont.)

Presupunem că ultima regulă a fost $(\wedge i)$. Aceasta înseamnă că

$$\varphi = \psi \wedge \chi$$

1	φ_1	premisa	Se observă că secvenții
	:		$\varphi_1, \dots, \varphi_n \vdash \psi$ și
n	φ_n	premisa	$\varphi_1, \dots, \varphi_n \vdash \chi$
	:		au demonstrații de lungime $< k$.
k_1	ψ		Din ipoteza de inducție rezultă
	:		$\varphi_1, \dots, \varphi_n \models \psi$ și
k_2	χ		$\varphi_1, \dots, \varphi_n \models \chi$
k	$\psi \wedge \chi$	$(\wedge i)k_1, k_2$	

Corectitudinea DN

Demonstrație (cont.)

Presupunem că ultima regulă a fost $(\wedge i)$. Aceasta înseamnă că

$$\varphi = \psi \wedge \chi$$

1	φ_1	<i>premisa</i>	Se observă că secvenții
	:		$\varphi_1, \dots, \varphi_n \vdash \psi$ și
n	φ_n	<i>premisa</i>	$\varphi_1, \dots, \varphi_n \vdash \chi$
	:		au demonstrații de lungime $< k$.
k_1	ψ		Din ipoteza de inducție rezultă
	:		$\varphi_1, \dots, \varphi_n \models \psi$ și
k_2	χ		$\varphi_1, \dots, \varphi_n \models \chi$ deci
k	$\psi \wedge \chi$	$(\wedge i)k_1, k_2$	$\varphi_1, \dots, \varphi_n \models \psi \wedge \chi$

Corectitudinea DN

Demonstrație (cont.)

Presupunem că ultima regulă a fost $(\rightarrow i)$. Aceasta înseamnă că

$$\varphi = \psi \rightarrow \chi$$

și ca în demonstrație există o cutie.

Corectitudinea DN

Demonstrație (cont.)

Presupunem că ultima regulă a fost $(\rightarrow i)$. Aceasta înseamnă că

$$\varphi = \psi \rightarrow \chi$$

și ca în demonstrație există o cutie.

1 φ_1 premisa

⋮

n φ_n premisa

⋮

k_1 ψ ipoteza

⋮

k_2 χ

k $\psi \rightarrow \chi$ $(\rightarrow i)k_1-k_2$

Corectitudinea DN

Demonstrație (cont.)

Presupunem că ultima regulă a fost $(\rightarrow i)$. Aceasta înseamnă că

$$\varphi = \psi \rightarrow \chi$$

și ca în demonstrație există o cutie.

1 φ_1 *premisa*

⋮

n φ_n *premisa*

⋮

k_1 ψ *ipoteza*

⋮

k_2 χ

Se observă că

$$\varphi_1, \dots, \varphi_n, \psi \vdash \chi$$

are demonstrația de lungime $< k$.

k $\psi \rightarrow \chi$ $(\rightarrow i)k_1 - k_2$

Corectitudinea DN

Demonstrație (cont.)

Presupunem că ultima regulă a fost $(\rightarrow i)$. Aceasta înseamnă că

$$\varphi = \psi \rightarrow \chi$$

și ca în demonstrație există o cutie.

1 φ_1 *premisa*

⋮

n φ_n *premisa*

⋮

k_1 ψ *ipoteza*

⋮

k_2 χ

Se observă că

$$\varphi_1, \dots, \varphi_n, \psi \vdash \chi$$

are demonstrația de lungime $< k$.

Din ipoteza de inducție rezultă

$$\varphi_1, \dots, \varphi_n, \psi \vdash \chi \quad (*)$$

k $\psi \rightarrow \chi \quad (\rightarrow i)_{k_1-k_2}$

Corectitudinea DN

Demonstrație (cont.)

Putem acum să demonstrăm că $\varphi_1, \dots, \varphi_n, \models \varphi$.

Corectitudinea DN

Demonstrație (cont.)

Putem acum să demonstrăm că $\varphi_1, \dots, \varphi_n, \models \varphi$.

Fie $e : Var \rightarrow \{0, 1\}$ o evaluare astfel încât $e^+(\varphi_1) = \dots = e^+(\varphi_n) = 1$.
Vrem să arătăm că $e^+(\varphi) = 1$.

Deoarece $\varphi = \psi \rightarrow \chi$ considerăm două cazuri.

Dacă $e^+(\psi) = 0$ atunci $e^+(\varphi) = 0 \rightarrow e^+(\chi) = 1$.

Corectitudinea DN

Demonstrație (cont.)

Putem acum să demonstrăm că $\varphi_1, \dots, \varphi_n, \models \varphi$.

Fie $e : Var \rightarrow \{0, 1\}$ o evaluare astfel încât $e^+(\varphi_1) = \dots = e^+(\varphi_n) = 1$.

Vrem să arătăm că $e^+(\varphi) = 1$.

Deoarece $\varphi = \psi \rightarrow \chi$ considerăm două cazuri.

Dacă $e^+(\psi) = 0$ atunci $e^+(\varphi) = 0 \rightarrow e^+(\chi) = 1$.

Dacă $e^+(\psi) = 1$ atunci e^+ este un model pentru formulele $\varphi_1, \dots, \varphi_n, \psi$.

Din (*) rezultă ca $e^+(\chi) = 1$, deci $e^+(\varphi) = 1 \rightarrow 1 = 1$.

Corectitudinea DN

Demonstrație (cont.)

Putem acum să demonstrăm că $\varphi_1, \dots, \varphi_n, \models \varphi$.

Fie $e : Var \rightarrow \{0, 1\}$ o evaluare astfel încât $e^+(\varphi_1) = \dots = e^+(\varphi_n) = 1$.

Vrem să arătăm că $e^+(\varphi) = 1$.

Deoarece $\varphi = \psi \rightarrow \chi$ considerăm două cazuri.

Dacă $e^+(\psi) = 0$ atunci $e^+(\varphi) = 0 \rightarrow e^+(\chi) = 1$.

Dacă $e^+(\psi) = 1$ atunci e^+ este un model pentru formulele $\varphi_1, \dots, \varphi_n, \psi$.

Din (*) rezultă ca $e^+(\chi) = 1$, deci $e^+(\varphi) = 1 \rightarrow 1 = 1$.

Am demonstrat că regula $(\rightarrow i)$ este corectă.

Corectitudinea DN

Demonstrație (cont.)

Putem acum să demonstrăm că $\varphi_1, \dots, \varphi_n, \models \varphi$.

Fie $e : Var \rightarrow \{0, 1\}$ o evaluare astfel încât $e^+(\varphi_1) = \dots = e^+(\varphi_n) = 1$.

Vrem să arătăm că $e^+(\varphi) = 1$.

Deoarece $\varphi = \psi \rightarrow \chi$ considerăm două cazuri.

Dacă $e^+(\psi) = 0$ atunci $e^+(\varphi) = 0 \rightarrow e^+(\chi) = 1$.

Dacă $e^+(\psi) = 1$ atunci e^+ este un model pentru formulele $\varphi_1, \dots, \varphi_n, \psi$.

Din (*) rezultă ca $e^+(\chi) = 1$, deci $e^+(\varphi) = 1 \rightarrow 1 = 1$.

Am demonstrat că regula $(\rightarrow i)$ este corectă.

Pentru a finaliza demonstrația trebuie să arătăm că fiecare din celelalte reguli ale deductiei naturale este corectă. □

PL - Deducție naturală: Completitudinea (optional)

Completitudinea DN (optional)

Teoremă

Deductia naturală este completă, i.e.

dacă $\varphi_1, \dots, \varphi_n \models \varphi$ atunci $\varphi_1, \dots, \varphi_n \vdash \varphi$ este valid

oricare ar fi formulele $\varphi_1, \dots, \varphi_n, \varphi$.

Notății

Pentru a demonstra ca DN este completă pentru PL facem urmatoarele notații:

Notări

Pentru a demonstra că DN este completă pentru PL facem urmatoarele notări:

- Fie $e : \text{Var} \rightarrow \{0, 1\}$ evaluare. Pentru orice $v \in \text{Var}$ definim

$$v^e := \begin{cases} v & \text{dacă } e(v) = 1 \\ \neg v & \text{dacă } e(v) = 0 \end{cases}$$

- $\text{Var}(\varphi) := \{v \in \text{Var} \mid v \text{ apare în } \varphi\}$ oricare φ formulă.

Completitudinea DN - rezultate ajutătoare

Propozitia 1

Fie φ este o formulă și $Var(\varphi) = \{v_1, \dots, v_n\}$. Pentru orice evaluare $e : Var \rightarrow \{0, 1\}$ sunt adevărate:

- $e^+(\varphi) = 1$ implica $\{v_1^e, \dots, v_n^e\} \vdash \varphi$ este valid,
- $e^+(\varphi) = 0$ implica $\{v_1^e, \dots, v_n^e\} \vdash \neg\varphi$ este valid.

Completitudinea DN - rezultate ajutătoare

Propozitia 1

Fie φ este o formulă și $Var(\varphi) = \{v_1, \dots, v_n\}$. Pentru orice evaluare $e : Var \rightarrow \{0, 1\}$ sunt adevărate:

- $e^+(\varphi) = 1$ implica $\{v_1^e, \dots, v_n^e\} \vdash \varphi$ este valid,
- $e^+(\varphi) = 0$ implica $\{v_1^e, \dots, v_n^e\} \vdash \neg\varphi$ este valid.

Propozitia 2

Oricare ar fi formulele $\varphi_1, \dots, \varphi_n, \varphi$,
daca $\varphi_1, \dots, \varphi_n \models \varphi$ atunci $\models \varphi_1 \rightarrow (\varphi_2 \rightarrow (\dots \rightarrow (\varphi_n \rightarrow \varphi) \dots))$.

Completitudinea DN - rezultate ajutătoare

Propozitia 1

Fie φ este o formulă și $Var(\varphi) = \{v_1, \dots, v_n\}$. Pentru orice evaluare $e : Var \rightarrow \{0, 1\}$ sunt adevărate:

- $e^+(\varphi) = 1$ implica $\{v_1^e, \dots, v_n^e\} \vdash \varphi$ este valid,
- $e^+(\varphi) = 0$ implica $\{v_1^e, \dots, v_n^e\} \vdash \neg\varphi$ este valid.

Propozitia 2

Oricare ar fi formulele $\varphi_1, \dots, \varphi_n, \varphi$,
dacă $\varphi_1, \dots, \varphi_n \models \varphi$ atunci $\models \varphi_1 \rightarrow (\varphi_2 \rightarrow (\dots \rightarrow (\varphi_n \rightarrow \varphi) \dots))$.

Propozitia 3

Oricare ar fi formulele $\varphi_1, \dots, \varphi_n, \varphi$,
dacă $\vdash \varphi_1 \rightarrow (\varphi_2 \rightarrow (\dots \rightarrow (\varphi_n \rightarrow \varphi) \dots))$ este valid,
atunci $\varphi_1, \dots, \varphi_n \vdash \varphi$ este valid.

Completitudinea DN

Teoremă

Deduçția naturală este completă, i.e.

dacă $\varphi_1, \dots, \varphi_n \models \varphi$ atunci $\varphi_1, \dots, \varphi_n \vdash \varphi$ este valid

oricare ar fi formulele $\varphi_1, \dots, \varphi_n, \varphi$.

Completitudinea DN

Teoremă

Deduçția naturală este completă, i.e.

dacă $\varphi_1, \dots, \varphi_n \models \varphi$ atunci $\varphi_1, \dots, \varphi_n \vdash \varphi$ este valid

oricare ar fi formulele $\varphi_1, \dots, \varphi_n, \varphi$.

Demonstratie

Pasul 1. Daca $\models \varphi$ atunci $\vdash \varphi$ este valid.

Completitudinea DN

Teoremă

Deduclia naturală este completă, i.e.

dacă $\varphi_1, \dots, \varphi_n \models \varphi$ atunci $\varphi_1, \dots, \varphi_n \vdash \varphi$ este valid

oricare ar fi formulele $\varphi_1, \dots, \varphi_n, \varphi$.

Demonstratie

Pasul 1. Daca $\models \varphi$ atunci $\vdash \varphi$ este valid.

Pasul 2. Presupunem ca $\varphi_1, \dots, \varphi_n \models \varphi$.

Completitudinea DN

Teoremă

Deduçția naturală este completă, i.e.

dacă $\varphi_1, \dots, \varphi_n \models \varphi$ atunci $\varphi_1, \dots, \varphi_n \vdash \varphi$ este valid

oricare ar fi formulele $\varphi_1, \dots, \varphi_n, \varphi$.

Demonstratie

Pasul 1. Daca $\models \varphi$ atunci $\vdash \varphi$ este valid.

Pasul 2. Presupunem ca $\varphi_1, \dots, \varphi_n \models \varphi$.

Din Propozitia 2 deducem ca $\vdash \varphi_1 \rightarrow (\varphi_2 \rightarrow (\dots \rightarrow (\varphi_n \rightarrow \varphi) \dots))$.

Completitudinea DN

Teoremă

Deducția naturală este completă, i.e.

dacă $\varphi_1, \dots, \varphi_n \models \varphi$ atunci $\varphi_1, \dots, \varphi_n \vdash \varphi$ este valid

oricare ar fi formulele $\varphi_1, \dots, \varphi_n, \varphi$.

Demonstratie

Pasul 1. Daca $\models \varphi$ atunci $\vdash \varphi$ este valid.

Pasul 2. Presupunem ca $\varphi_1, \dots, \varphi_n \models \varphi$.

Din Propozitia 2 deducem ca $\vdash \varphi_1 \rightarrow (\varphi_2 \rightarrow (\dots \rightarrow (\varphi_n \rightarrow \varphi) \dots))$.

Aplicand Pasul 1 obtinem ca $\vdash \varphi_1 \rightarrow (\varphi_2 \rightarrow (\dots \rightarrow (\varphi_n \rightarrow \varphi) \dots))$ este valid.

Completitudinea DN

Teoremă

Deducția naturală este completă, i.e.

dacă $\varphi_1, \dots, \varphi_n \models \varphi$ atunci $\varphi_1, \dots, \varphi_n \vdash \varphi$ este valid

oricare ar fi formulele $\varphi_1, \dots, \varphi_n, \varphi$.

Demonstratie

Pasul 1. Daca $\models \varphi$ atunci $\vdash \varphi$ este valid.

Pasul 2. Presupunem ca $\varphi_1, \dots, \varphi_n \models \varphi$.

Din Propozitia 2 deducem ca $\vdash \varphi_1 \rightarrow (\varphi_2 \rightarrow (\dots \rightarrow (\varphi_n \rightarrow \varphi) \dots))$.

Aplicand Pasul 1 obtinem ca $\vdash \varphi_1 \rightarrow (\varphi_2 \rightarrow (\dots \rightarrow (\varphi_n \rightarrow \varphi) \dots))$ este valid. In consecinta $\varphi_1, \dots, \varphi_n \vdash \varphi$ este valid din Propozitia 3.

Completitudinea DN

Demonstratie (cont.)

În continuare demonstrează Pasul 1.

Fie φ o tautologie, i.e. $\models \varphi$, astfel încât $Var(\varphi) = \{p_1, \dots, p_n\}$.

Completitudinea DN

Demonstratie (cont.)

În continuare demonstrează Pasul 1.

Fie φ o tautologie, i.e. $\models \varphi$, astfel încât $Var(\varphi) = \{p_1, \dots, p_n\}$.

Oricare ar fi $e : Var \rightarrow \{0, 1\}$ stim că $e^+(\varphi) = 1$ deci, din Propozitia 1, rezultă că secventul $\{p_1^e, \dots, p_n^e\} \vdash \varphi$ este valid.

Completitudinea DN

Demonstratie (cont.)

În continuare demonstrează Pasul 1.

Fie φ o tautologie, i.e. $\models \varphi$, astfel încât $\text{Var}(\varphi) = \{p_1, \dots, p_n\}$.

Oricare ar fi $e : \text{Var} \rightarrow \{0, 1\}$ stim că $e^+(\varphi) = 1$ deci, din Propozitia 1, rezultă că seceventul $\{p_1^e, \dots, p_n^e\} \vdash \varphi$ este valid.

Deoarece există 2^n evaluări, i.e., tabelul de adevar are 2^n linii, obținem 2^n demonstrații pentru φ , fiecare din aceste demonstrații având n premise.

Completitudinea DN

Demonstratie (cont.)

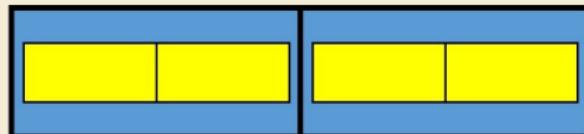
În continuare demonstram Pasul 1.

Fie φ o tautologie, i.e. $\models \varphi$, astfel incat $Var(\varphi) = \{p_1, \dots, p_n\}$.

Oricare ar fi $e : Var \rightarrow \{0, 1\}$ stim ca $e^+(\varphi) = 1$ deci, din Propozitia 1, rezulta ca seceventul $\{p_1^e, \dots, p_n^e\} \vdash \varphi$ este valid.

Deoarece exista 2^n evaluari, i.e., tabelul de adevar are 2^n linii, obtinem 2^n demonstratii pentru φ , fiecare din aceste demonstratii avand n premise.

Vom arata in continuare, pe un exemplu simplu, cum se pot combina aceste 2^n demonstratii cu premise pentru a obtine o demonstratie fara premise pentru φ .



Completitudinea DN

Demonstratie (cont.)

Consideram $\models \varphi$ si $n = 2$, i.e. $\text{Var}(\varphi) = \{p_1, p_2\}$.

De exemplu, puteti considera $\varphi = p_1 \wedge p_2 \rightarrow p_1$

Completitudinea DN

Demonstratie (cont.)

Consideram $\models \varphi$ si $n = 2$, i.e. $\text{Var}(\varphi) = \{p_1, p_2\}$.

De exemplu, puteti considera $\varphi = p_1 \wedge p_2 \rightarrow p_1$

Din Propozitia 1 stim ca urmatorii sevenți sunt valizi:

$$\begin{array}{ll} p_1, p_2 & \vdash \varphi \\ p_1, \neg p_2 & \vdash \varphi \\ \neg p_1, p_2 & \vdash \varphi \\ \neg p_1, \neg p_2 & \vdash \varphi \end{array}$$

Completitudinea DN

Demonstratie (cont.)

Consideram $\models \varphi$ si $n = 2$, i.e. $\text{Var}(\varphi) = \{p_1, p_2\}$.

De exemplu, puteti considera $\varphi = p_1 \wedge p_2 \rightarrow p_1$

Din Propozitia 1 stim ca urmatorii sevenți sunt valizi:

$$\begin{array}{ll} p_1, p_2 & \vdash \varphi \\ p_1, \neg p_2 & \vdash \varphi \\ \neg p_1, p_2 & \vdash \varphi \\ \neg p_1, \neg p_2 & \vdash \varphi \end{array}$$

deci există demonstratiile:

p_1	ipoteza
p_2	ipoteza
:	
φ	

p_1	ipoteza
$\neg p_2$	ipoteza
:	
φ	

$\neg p_1$	ipoteza
p_2	ipoteza
:	
φ	

$\neg p_1$	ipoteza
$\neg p_2$	ipoteza
:	
φ	

Completitudinea DN

Demonstratie (cont.)

Combinam cele patru demonstratii astfel:

Completitudinea DN

Demonstratie (cont.)

Combinam cele patru demonstratii astfel:

$$\frac{p_1 \vee \neg p_1}{\boxed{\begin{array}{ll} p_1 & \text{ipoteza} \end{array}}} \quad \boxed{\begin{array}{ll} \neg p_1 & \text{ipoteza} \end{array}} \quad TND$$

Completitudinea DN

Demonstratie (cont.)

Combinam cele patru demonstratii astfel:

$$p_1 \vee \neg p_1$$

p_1	<i>ipoteza</i>
$p_2 \vee \neg p_2$	<i>TND</i>
p_2	<i>ipoteza</i>
$\neg p_2$	<i>ipoteza</i>

TND

$\neg p_1$	<i>ipoteza</i>
$p_2 \vee \neg p_2$	<i>TND</i>
p_2	<i>ipoteza</i>
$\neg p_2$	<i>ipoteza</i>

Completitudinea DN

Demonstratie (cont.)

Combinam cele patru demonstratii astfel:

$$p_1 \vee \neg p_1$$

$$p_1$$

ipoteza

TND

$$p_2 \vee \neg p_2$$

$$p_2 \quad \text{ipoteza}$$

:

$$\varphi$$

$$\varphi$$

$$\neg p_2 \quad \text{ipoteza}$$

:

$$\varphi$$

($\vee e$)

TND

$$\neg p_1$$

ipoteza

TND

$$p_2 \vee \neg p_2$$

$$p_2 \quad \text{ipoteza}$$

:

$$\varphi$$

$$\varphi$$

($\vee e$)

Completitudinea DN

Demonstratie (cont.)

Combinam cele patru demonstratii astfel:

$$p_1 \vee \neg p_1$$

p_1		<i>ipoteza</i>
$p_2 \vee \neg p_2$		<i>TND</i>
p_2	<i>ipoteza</i>	$\neg p_2$
:		:
φ		φ
		(ve)

TND

ipoteza

TND

$$\neg p_1$$

$$p_2 \vee \neg p_2$$

p_2 *ipoteza*

:

φ

φ

(ve)

ipoteza

TND

$\neg p_2$ *ipoteza*

:

φ

φ

(ve)

Completitudinea DN

Demonstratie (cont.)

Combinam cele patru demonstratii astfel:

$$p_1 \vee \neg p_1$$

p_1		<i>ipoteza</i>
$p_2 \vee \neg p_2$		<i>TND</i>
p_2	<i>ipoteza</i>	
:		
φ		
φ		($\vee e$)

TND

$$\neg p_1$$

$\neg p_1$		<i>ipoteza</i>
$p_2 \vee \neg p_2$		<i>TND</i>
p_2	<i>ipoteza</i>	
:		
φ		
φ		($\vee e$)

ipoteza

TND

Am obtinut o demonstratie pentru φ fara ipoteze.

□

Deductia naturala DN

- este un sistem deductiv corect si complet pentru logica clasica,
- stabileste reguli de deductie pentru fiecare operator logic,
- o demonstratie se construieste prin aplicarea succesiva a regulilor de deductie,
- in demonstratii putem folosi ipoteze temporare, scopul acestora fiind bine delimitat.

Pe săptămâna viitoare!

Curs 6

2021-2022

Fundamentele Limbajelor de Programare

Cuprins

- 1** Clauze propoziționale definite
- 2** Puncte fixe. Teorema Knaster-Tarski
- 3** Completitudinea sistemului de deducție CDP

Problema satisfiabilității

- În principiu, putem verifica problema consecinței logice construind un **tabel de adevăr**, cu câte o linie pentru fiecare interpretare posibilă.
- În cazul în care programul și ținta conțin n atomi diferiți, tabelul de adevăr rezultat o să aibă 2^n rânduri.
- Această metodă este extrem costisitoare computațional (**timp exponențial**).

Cum salvăm situația?

Problema satisfiabilității

- În principiu, putem verifica problema consecinței logice construind un **tabel de adevăr**, cu câte o linie pentru fiecare interpretare posibilă.
- În cazul în care programul și ținta conțin n atomi diferiți, tabelul de adevăr rezultat o să aibă 2^n rânduri.
- Această metodă este extrem costisitoare computațional (**timp exponențial**).

Cum salvăm situația?

- 1 Folosirea **metodelor sintactice** pentru a stabili problema consecinței logice (*proof search*)
- 2 **Restricționarea formulelor** din "programele logice" (**clauze definite**)

Clauze propoziționale definite

Clauze propoziționale definite

□ O clauză propozițională definită este o formulă care poate avea una din formele:

- 1 q
- 2 $p_1 \wedge \dots \wedge p_k \rightarrow q$

unde q, p_1, \dots, p_n sunt variabile propoziționale

Clauze propoziționale definite

- O clauză propozițională definită este o formulă care poate avea una din formele:

- 1 q (un fapt în Prolog $q.$)
- 2 $p_1 \wedge \dots \wedge p_k \rightarrow q$ (o regulă în Prolog $q :- p_1, \dots, p_k$)

unde q, p_1, \dots, p_n sunt variabile propoziționale

Clauze propoziționale definite

- O clauză propozițională definită este o formulă care poate avea una din formele:

1 q (un fapt în Prolog $q.$)
2 $p_1 \wedge \dots \wedge p_k \rightarrow q$ (o regulă în Prolog $q :- p_1, \dots, p_k$)

unde q, p_1, \dots, p_n sunt variabile propoziționale

- Numim variabilele propoziționale atomi.

Clauze propoziționale definite

- O clauză propozițională definită este o formulă care poate avea una din formele:
 - 1 q (un fapt în Prolog `q.`)
 - 2 $p_1 \wedge \dots \wedge p_k \rightarrow q$ (o regulă în Prolog `q :- p1, ..., pk.`)unde q, p_1, \dots, p_n sunt variabile propoziționale
- Numim variabilele propoziționale atomi.

Programare logică – cazul logicii propoziționale

- Un "program logic" este o listă Cd_1, \dots, Cd_n de clauze definite.

Clauze propoziționale definite

- O clauză propozițională definită este o formulă care poate avea una din formele:
 - 1 q (un fapt în Prolog $q.$)
 - 2 $p_1 \wedge \dots \wedge p_k \rightarrow q$ (o regulă în Prolog $q :- p_1, \dots, p_k$)
- unde q, p_1, \dots, p_n sunt variabile propoziționale
- Numim variabilele propoziționale atomi.

Programare logică – cazul logicii propoziționale

- Un "program logic" este o listă Cd_1, \dots, Cd_n de clauze definite.
- O întrebare este o listă q_1, \dots, q_m de atomi.

Clauze propoziționale definite

- O clauză propozițională definită este o formulă care poate avea una din formele:
 - 1 q (un fapt în Prolog `q.`)
 - 2 $p_1 \wedge \dots \wedge p_k \rightarrow q$ (o regulă în Prolog `q :- p1, ..., pk.`)unde q, p_1, \dots, p_n sunt variabile propoziționale
- Numim variabilele propoziționale atomi.

Programare logică – cazul logicii propoziționale

- Un "program logic" este o listă Cd_1, \dots, Cd_n de clauze definite.
- O întrebare este o listă q_1, \dots, q_m de atomi.
- Sarcina sistemului este să stabilească:

$$Cd_1, \dots, Cd_n \models q_1 \wedge \dots \wedge q_m.$$

Clauze propoziționale definite

- O clauză propozițională definită este o formulă care poate avea una din formele:
 - 1 q (un fapt în Prolog `q.`)
 - 2 $p_1 \wedge \dots \wedge p_k \rightarrow q$ (o regulă în Prolog `q :- p1, ..., pk.`)
- unde q, p_1, \dots, p_n sunt variabile propoziționale
- Numim variabilele propoziționale atomi.

Programare logică – cazul logicii propoziționale

- Un "program logic" este o listă Cd_1, \dots, Cd_n de clauze definite.
- O întrebare este o listă q_1, \dots, q_m de atomi.
- Sarcina sistemului este să stabilească:

$$Cd_1, \dots, Cd_n \models q_1 \wedge \dots \wedge q_m.$$

Vom studia metode sintactice pentru a rezolva această problemă!

Sistem de deducție CDP

Sistem de deducție CDP pentru clauze definite propoziționale

Pentru o mulțime S de clauze definite propoziționale, avem

Sistem de deducție CDP

Sistem de deducție CDP pentru clauze definite propoziționale

Pentru o mulțime \mathcal{S} de clauze definite propoziționale, avem

- Axiome** (premise): orice clauză din \mathcal{S}

Sistem de deducție CDP

Sistem de deducție CDP pentru clauze definite propoziționale

Pentru o mulțime \mathcal{S} de clauze definite propoziționale, avem

- Axiome (premise): orice clauză din \mathcal{S}
- Reguli de deducție:

$$\frac{P \quad P \rightarrow Q}{Q} (MP) \qquad \frac{P \quad Q}{P \wedge Q} (andI)$$

- Aceste reguli ne permit să deducem formula de sub linie din formulele de deasupra liniei.
- Sunt regulile ($\rightarrow e$) și ($\wedge i$) din deducția naturală pentru logica propozițională.

Sistemul de deducție CDP

$$\frac{P \quad P \rightarrow Q}{Q} \text{ (MP)}$$

$$\frac{P \quad Q}{P \wedge Q} \text{ (andI)}$$

Exemplu

oslo	→	windy
oslo	→	norway
norway	→	cold
cold \wedge windy	→	winterIsComing
		oslo

Sistemul de deducție CDP

$$\frac{P \quad P \rightarrow Q}{Q} \text{ (MP)} \qquad \frac{P \quad Q}{P \wedge Q} \text{ (andI)}$$

Exemplu

```

oslo      →   windy
oslo      →   norway
norway    →   cold
cold ∧ windy →   winterIsComing
                           oslo

```

<i>oslo</i>	<i>oslo</i> → <i>norway</i>	<i>norway</i> → <i>cold</i>	<i>oslo</i>	<i>oslo</i> → <i>windy</i>
	<i>norway</i>			<i>windy</i>
<i>cold</i>				
<i>cold</i> ∧ <i>windy</i>				

Sistemul de deducție CDP

$$\frac{P \quad P \rightarrow Q}{Q} \quad (MP)$$

$$\frac{P \quad Q}{P \wedge Q} \text{ (andI)}$$

Exemplu

```

oslo      →    windy
oslo      →    norway
norway   →    cold
cold ∧ windy →    winterIsComing
                           oslo

```

<i>oslo</i>	<i>oslo</i> → <i>norway</i>	<i>norway</i> → <i>cold</i>	<i>oslo</i>	<i>oslo</i> → <i>windy</i>
	<i>norway</i>			<i>windy</i>
<i>cold</i>				

$$\frac{\text{cold} \wedge \text{windy}}{\text{winterIsComing}}$$

Sistemul de deducție CDP

$$\frac{P \quad P \rightarrow Q}{Q} \text{ (MP)}$$

$$\frac{P \quad Q}{P \wedge Q} \text{ (andI)}$$

Exemplu

oslo	→	windy
oslo	→	norway
norway	→	cold
cold \wedge windy	→	winterIsComing
		oslo

1. *oslo* → *windy*
2. *oslo* → *norway*
3. *norway* → *cold*
4. *cold* \wedge *windy* → *winterIsComing*
5. *oslo*

6. *norway* (MP 5,2)
7. *cold* (MP 6,3)
8. *windy* (MP 5,1)
9. *cold* \wedge *windy* (andI 7,8)
10. *winterIsComing* (MP 9,4)

Sistemul de deducție CDP

O formulă Q se poate deduce din S în sistemul de deducție CDP, notat

$$S \vdash Q,$$

dacă există o secvență de formule Q_1, \dots, Q_n astfel încât $Q_n = Q$ și fiecare Q_i :

- fie aparține lui S
- fie se poate deduce din Q_1, \dots, Q_{i-1} folosind regulile de deducție (*MP*) și (*andI*)

Sistemul de deducție CDP

Putem folosi sistemul de deducție CDP pentru a deduce alți atomi:

Sistemul de deducție CDP

Putem folosi sistemul de deducție CDP pentru a deduce alți atomi:

- Atomii $p_i \in \mathcal{S}$ care sunt fapte sunt deductibili.
 - Sunt deduși ca axiome.

Sistemul de deducție CDP

Putem folosi sistemul de deducție CDP pentru a deduce alți atomi:

- Atomii $p_i \in \mathcal{S}$ care sunt fapte sunt deductibili.
 - Sunt deduși ca axiome.
- Un atom r este deductibil dacă
 - p_1, \dots, p_n sunt deductibili, și
 - $p_1 \wedge \dots \wedge p_n \rightarrow r$ este în \mathcal{S} .

O astfel de derivare folosește de $n - 1$ ori (*andI*) și o data (*MP*).

Sistemul de deducție CDP

Putem folosi sistemul de deducție CDP pentru a deduce alți atomi:

- Atomii $p_i \in \mathcal{S}$ care sunt fapte sunt deductibili.
 - Sunt deduși ca axiome.
- Un atom r este deductibil dacă
 - p_1, \dots, p_n sunt deductibili, și
 - $p_1 \wedge \dots \wedge p_n \rightarrow r$ este în \mathcal{S} .

O astfel de derivare folosește de $n - 1$ ori (*andI*) și o data (*MP*).

Deci putem construi mulțimi din ce în ce mai mari de atomi care sunt consecințe logice din \mathcal{S} , și pentru care există derivări din \mathcal{S} .

Sistemul de deducție CDP

Putem folosi sistemul de deducție CDP pentru a deduce alți atomi:

- Atomii $p_i \in \mathcal{S}$ care sunt fapte sunt deductibili.
 - Sunt deduși ca axiome.
- Un atom r este deductibil dacă
 - p_1, \dots, p_n sunt deductibili, și
 - $p_1 \wedge \dots \wedge p_n \rightarrow r$ este în \mathcal{S} .

O astfel de derivare folosește de $n - 1$ ori (*andI*) și o data (*MP*).

Deci putem construi mulțimi din ce în ce mai mari de atomi care sunt consecințe logice din \mathcal{S} , și pentru care există derivări din \mathcal{S} .

Observăm că (*andI*) și (*MP*) pot fi înlocuite cu următoarea regulă derivată:

$$\frac{P_1, \dots, P_n \quad P_1 \wedge \dots \wedge P_n \rightarrow Q}{Q} \text{ (GMP)}$$

Completitudinea sistemului de deducție CDP

- Se poate demonstra că aceste **reguli sunt corecte**, folosind tabelele de adevăr.
 - Dacă formulele de deasupra liniei sunt adevărate, atunci și formula de sub linie este adevărată.

Compleitudinea sistemului de deducție CDP

- Se poate demonstra că aceste **reguli sunt corecte**, folosind tabelele de adevăr.
 - Dacă formulele de deasupra liniei sunt adevărate, atunci și formula de sub linie este adevărată.
- Mai mult, **sistemul de deducție este și complet**, adică
 - dacă $\mathcal{S} \models q$, atunci $\mathcal{S} \vdash q$.
 - Dacă q este o consecință logică a lui \mathcal{S} , atunci există o derivare a sa din \mathcal{S} folosind sistemul de deducție CDP

Completitudinea sistemului de deducție CDP

- Se poate demonstra că aceste **reguli sunt corecte**, folosind tabelele de adevăr.
 - Dacă formulele de deasupra liniei sunt adevărate, atunci și formula de sub linie este adevărată.
- Mai mult, **sistemul de deducție este și complet**, adică
 - dacă $\mathcal{S} \models q$, atunci $\mathcal{S} \vdash q$.
 - Dacă q este o consecință logică a lui \mathcal{S} , atunci există o derivare a sa din \mathcal{S} folosind sistemul de deducție CDP
- Pentru a demonstra completitudinea vom folosi **teorema Knaster-Tarski**.

Puncte fixe. Teorema Knaster-Tarski

Mulțimi parțial ordonate

- O mulțime parțial ordonată (mpo) este o pereche (M, \leq) unde $\leq \subseteq M \times M$ este o relație de ordine.
 - relație de ordine: reflexivă, antisimetrică, tranzitivă

Mulțimi parțial ordonate

- O mulțime parțial ordonată (mpo) este o pereche (M, \leq) unde $\leq \subseteq M \times M$ este o relație de ordine.
 - relație de ordine: reflexivă, antisimetrică, tranzitivă
- O mpo (L, \leq) se numește lanț dacă este total ordonată, adică $x \leq y$ sau $y \leq x$ pentru orice $x, y \in L$. Vom considera lanțuri numărabile:

$$x_1 \leq x_2 \leq x_3 \leq \dots$$

Mulțimi parțial ordonate complete

O mpo (C, \leq) este completă (cpo) dacă:

- C are prim element \perp ($\perp \leq x$ oricare $x \in C$),
- $\bigvee_n x_n$ există în C pentru orice lanț $x_1 \leq x_2 \leq x_3 \leq \dots$

Mulțimi parțial ordonate complete

O mpo (C, \leq) este completă (cpo) dacă:

- C are prim element \perp ($\perp \leq x$ oricare $x \in C$),
- $\bigvee_n x_n$ există în C pentru orice lanț $x_1 \leq x_2 \leq x_3 \leq \dots$

Exemplu

Fie X o mulțime și $\mathcal{P}(X)$ mulțimea submulțimilor lui X .

$(\mathcal{P}(X), \subseteq)$ este o cpo:

- \subseteq este o relație de ordine
- \emptyset este prim element ($\emptyset \subseteq Q$ pentru orice $Q \in \mathcal{P}(X)$)
- pentru orice sir (numărabil) de submulțimi ale lui X $Q_1 \subseteq Q_2 \subseteq \dots$ evident $\bigcup_n Q_n \in \mathcal{P}(X)$

Funcție monotonă

□ Fie (A, \leq_A) și (B, \leq_B) mulțimi parțial ordonate.

O funcție $f : A \rightarrow B$ este **monotonă (crescătoare)**

dacă $a_1 \leq_A a_2$ implică $f(a_1) \leq_B f(a_2)$ oricare $a_1, a_2 \in A$.

Funcție monotonă

□ Fie (A, \leq_A) și (B, \leq_B) mulțimi parțial ordonate.

O funcție $f : A \rightarrow B$ este **monotonă (crescătoare)**

dacă $a_1 \leq_A a_2$ implică $f(a_1) \leq_B f(a_2)$ oricare $a_1, a_2 \in A$.

Exemplu

Fie următoarele funcții $f_i : \mathcal{P}(\{1, 2, 3\}) \rightarrow \mathcal{P}(\{1, 2, 3\})$ cu $i \in \{1, 2, 3\}$

□ $f_1(Y) = Y \cup \{1\}$

Funcție monotonă

□ Fie (A, \leq_A) și (B, \leq_B) mulțimi parțial ordonate.

O funcție $f : A \rightarrow B$ este **monotonă (crescătoare)**

dacă $a_1 \leq_A a_2$ implică $f(a_1) \leq_B f(a_2)$ oricare $a_1, a_2 \in A$.

Exemplu

Fie următoarele funcții $f_i : \mathcal{P}(\{1, 2, 3\}) \rightarrow \mathcal{P}(\{1, 2, 3\})$ cu $i \in \{1, 2, 3\}$

□ $f_1(Y) = Y \cup \{1\}$ este monotonă.

Funcție monotonă

□ Fie (A, \leq_A) și (B, \leq_B) mulțimi parțial ordonate.

O funcție $f : A \rightarrow B$ este **monotonă (crescătoare)**

dacă $a_1 \leq_A a_2$ implică $f(a_1) \leq_B f(a_2)$ oricare $a_1, a_2 \in A$.

Exemplu

Fie următoarele funcții $f_i : \mathcal{P}(\{1, 2, 3\}) \rightarrow \mathcal{P}(\{1, 2, 3\})$ cu $i \in \{1, 2, 3\}$

□ $f_1(Y) = Y \cup \{1\}$ este monotonă.

□ $f_2(Y) = \begin{cases} \{1\} & \text{dacă } 1 \in Y \\ \emptyset & \text{altfel} \end{cases}$

Funcție monotonă

□ Fie (A, \leq_A) și (B, \leq_B) mulțimi parțial ordonate.

O funcție $f : A \rightarrow B$ este **monotonă (crescătoare)**

dacă $a_1 \leq_A a_2$ implică $f(a_1) \leq_B f(a_2)$ oricare $a_1, a_2 \in A$.

Exemplu

Fie următoarele funcții $f_i : \mathcal{P}(\{1, 2, 3\}) \rightarrow \mathcal{P}(\{1, 2, 3\})$ cu $i \in \{1, 2, 3\}$

□ $f_1(Y) = Y \cup \{1\}$ este monotonă.

□ $f_2(Y) = \begin{cases} \{1\} & \text{dacă } 1 \in Y \\ \emptyset & \text{altfel} \end{cases}$ este monotonă.

Funcție monotonă

□ Fie (A, \leq_A) și (B, \leq_B) mulțimi parțial ordonate.

O funcție $f : A \rightarrow B$ este **monotonă (crescătoare)**

dacă $a_1 \leq_A a_2$ implică $f(a_1) \leq_B f(a_2)$ oricare $a_1, a_2 \in A$.

Exemplu

Fie următoarele funcții $f_i : \mathcal{P}(\{1, 2, 3\}) \rightarrow \mathcal{P}(\{1, 2, 3\})$ cu $i \in \{1, 2, 3\}$

□ $f_1(Y) = Y \cup \{1\}$ este monotonă.

□ $f_2(Y) = \begin{cases} \{1\} & \text{dacă } 1 \in Y \\ \emptyset & \text{altfel} \end{cases}$ este monotonă.

□ $f_3(Y) = \begin{cases} \emptyset & \text{dacă } 1 \in Y \\ \{1\} & \text{altfel} \end{cases}$

Funcție monotonă

□ Fie (A, \leq_A) și (B, \leq_B) mulțimi parțial ordonate.

O funcție $f : A \rightarrow B$ este **monotonă (crescătoare)**

dacă $a_1 \leq_A a_2$ implică $f(a_1) \leq_B f(a_2)$ oricare $a_1, a_2 \in A$.

Exemplu

Fie următoarele funcții $f_i : \mathcal{P}(\{1, 2, 3\}) \rightarrow \mathcal{P}(\{1, 2, 3\})$ cu $i \in \{1, 2, 3\}$

□ $f_1(Y) = Y \cup \{1\}$ este monotonă.

□ $f_2(Y) = \begin{cases} \{1\} & \text{dacă } 1 \in Y \\ \emptyset & \text{altfel} \end{cases}$ este monotonă.

□ $f_3(Y) = \begin{cases} \emptyset & \text{dacă } 1 \in Y \\ \{1\} & \text{altfel} \end{cases}$ nu este monotonă.

Funcție monotonă

□ Fie (A, \leq_A) și (B, \leq_B) mulțimi parțial ordonate.

O funcție $f : A \rightarrow B$ este **monotonă (crescătoare)**

dacă $a_1 \leq_A a_2$ implică $f(a_1) \leq_B f(a_2)$ oricare $a_1, a_2 \in A$.

Exemplu

Fie următoarele funcții $f_i : \mathcal{P}(\{1, 2, 3\}) \rightarrow \mathcal{P}(\{1, 2, 3\})$ cu $i \in \{1, 2, 3\}$

□ $f_1(Y) = Y \cup \{1\}$ este monotonă.

□ $f_2(Y) = \begin{cases} \{1\} & \text{dacă } 1 \in Y \\ \emptyset & \text{altfel} \end{cases}$ este monotonă.

□ $f_3(Y) = \begin{cases} \emptyset & \text{dacă } 1 \in Y \\ \{1\} & \text{altfel} \end{cases}$ nu este monotonă.

De exemplu, $\emptyset \subseteq \{1\}$, dar $f_3(\emptyset) = \{1\}$, $f_3(\{1\}) = \emptyset$ și $f_3(\emptyset) \not\subseteq f_3(\{1\})$.

Funcție continuă

□ Fie (A, \leq_A) și (B, \leq_B) mulțimi parțial ordonate complete.

O funcție $f : A \rightarrow B$ este **continuă** dacă

$$f(\bigvee_n a_n) = \bigvee_n f(a_n) \text{ pentru orice lanț } \{a_n\}_n \text{ din } A.$$

Funcție continuă

- Fie (A, \leq_A) și (B, \leq_B) mulțimi parțial ordonate complete.

O funcție $f : A \rightarrow B$ este **continuă** dacă

$$f(\bigvee_n a_n) = \bigvee_n f(a_n) \text{ pentru orice lanț } \{a_n\}_n \text{ din } A.$$

- Observăm că **orice funcție continuă este crescătoare**.

Funcție continuă

- Fie (A, \leq_A) și (B, \leq_B) mulțimi parțial ordonate complete.

O funcție $f : A \rightarrow B$ este **continuă** dacă

$$f(\bigvee_n a_n) = \bigvee_n f(a_n) \text{ pentru orice lanț } \{a_n\}_n \text{ din } A.$$

- Observăm că **orice funcție continuă este crescătoare**.

Exemplu

Fie următoarele funcții $f_i : \mathcal{P}(\{1, 2, 3\}) \rightarrow \mathcal{P}(\{1, 2, 3\})$ cu $i \in \{1, 2, 3\}$

- $f_1(Y) = Y \cup \{1\}$ este continuă.

Funcție continuă

- Fie (A, \leq_A) și (B, \leq_B) mulțimi parțial ordonate complete.

O funcție $f : A \rightarrow B$ este **continuă** dacă

$$f(\bigvee_n a_n) = \bigvee_n f(a_n) \text{ pentru orice lanț } \{a_n\}_n \text{ din } A.$$

- Observăm că **orice funcție continuă este crescătoare**.

Exemplu

Fie următoarele funcții $f_i : \mathcal{P}(\{1, 2, 3\}) \rightarrow \mathcal{P}(\{1, 2, 3\})$ cu $i \in \{1, 2, 3\}$

- $f_1(Y) = Y \cup \{1\}$ este continuă.

- $f_3(Y) = \begin{cases} \emptyset & \text{dacă } 1 \in Y \\ \{1\} & \text{altfel} \end{cases}$ nu este continuă.

Funcție continuă

- Fie (A, \leq_A) și (B, \leq_B) mulțimi parțial ordonate complete.

O funcție $f : A \rightarrow B$ este **continuă** dacă

$$f(\bigvee_n a_n) = \bigvee_n f(a_n) \text{ pentru orice lanț } \{a_n\}_n \text{ din } A.$$

- Observăm că **orice funcție continuă este crescătoare**.

Exemplu

Fie următoarele funcții $f_i : \mathcal{P}(\{1, 2, 3\}) \rightarrow \mathcal{P}(\{1, 2, 3\})$ cu $i \in \{1, 2, 3\}$

- $f_1(Y) = Y \cup \{1\}$ este continuă.

- $f_3(Y) = \begin{cases} \emptyset & \text{dacă } 1 \in Y \\ \{1\} & \text{altfel} \end{cases}$ nu este continuă.

De exemplu, consideram lantul $\emptyset \subseteq \{1\}$.

Avem $\emptyset \cup \{1\} = \{1\}$ și $f_3(\{1\}) = \emptyset$.

Dar $f_3(\emptyset) = \{1\}$, $f_3(\{1\}) = \emptyset$ și $f_3(\emptyset) \cup f_3(\{1\}) = \{1\}$.

Teorema de punct fix

- Un element $a \in C$ este **punct fix** al unei funcții $f : C \rightarrow C$ dacă $f(a) = a$.

Teorema de punct fix

- Un element $a \in C$ este **punct fix** al unei funcții $f : C \rightarrow C$ dacă $f(a) = a$.

Teorema Knaster-Tarski pentru CPO

Fie (C, \leq) o mulțime parțial ordonată completă și $\mathbf{F} : C \rightarrow C$ o funcție continuă. Atunci

$$a = \bigvee_n \mathbf{F}^n(\perp)$$

este cel mai mic punct fix al funcției \mathbf{F} .

Teorema de punct fix

- Un element $a \in C$ este **punct fix** al unei funcții $f : C \rightarrow C$ dacă $f(a) = a$.

Teorema Knaster-Tarski pentru CPO

Fie (C, \leq) o mulțime parțial ordonată completă și $\mathbf{F} : C \rightarrow C$ o funcție continuă. Atunci

$$a = \bigvee_n \mathbf{F}^n(\perp)$$

este cel mai mic punct fix al funcției \mathbf{F} .

- Observăm că în ipotezele ultimei teoreme secvența

$$\mathbf{F}^0(\perp) = \perp \leq \mathbf{F}(\perp) \leq \mathbf{F}^2(\perp) \leq \cdots \leq \mathbf{F}^n(\perp) \leq \cdots$$

este un lanț, deci $\bigvee_n \mathbf{F}^n(\perp)$ există.

Teorema Knaster-Tarski pentru CPO

Demonstrație

Fie (C, \leq) o mulțime parțial ordonată completă și $\mathbf{F} : C \rightarrow C$ o funcție continuă.

- Arătăm că $a = \bigvee_n \mathbf{F}^n(\perp)$ este punct fix, i.e. $\mathbf{F}(a) = a$

Teorema Knaster-Tarski pentru CPO

Demonstrație

Fie (C, \leq) o mulțime parțial ordonată completă și $\mathbf{F} : C \rightarrow C$ o funcție continuă.

- Arătăm că $a = \bigvee_n \mathbf{F}^n(\perp)$ este punct fix, i.e. $\mathbf{F}(a) = a$

$$\mathbf{F}(a) = \mathbf{F}(\bigvee_n \mathbf{F}^n(\perp))$$

Teorema Knaster-Tarski pentru CPO

Demonstrație

Fie (C, \leq) o mulțime parțial ordonată completă și $\mathbf{F} : C \rightarrow C$ o funcție continuă.

- Arătăm că $a = \bigvee_n \mathbf{F}^n(\perp)$ este punct fix, i.e. $\mathbf{F}(a) = a$

$$\begin{aligned}\mathbf{F}(a) &= \mathbf{F}(\bigvee_n \mathbf{F}^n(\perp)) \\ &= \bigvee_n \mathbf{F}(\mathbf{F}^n(\perp)) \text{ din continuitate}\end{aligned}$$

Teorema Knaster-Tarski pentru CPO

Demonstrație

Fie (C, \leq) o mulțime parțial ordonată completă și $\mathbf{F} : C \rightarrow C$ o funcție continuă.

- Arătăm că $a = \bigvee_n \mathbf{F}^n(\perp)$ este punct fix, i.e. $\mathbf{F}(a) = a$

$$\begin{aligned}\mathbf{F}(a) &= \mathbf{F}(\bigvee_n \mathbf{F}^n(\perp)) \\ &= \bigvee_n \mathbf{F}(\mathbf{F}^n(\perp)) \text{ din continuitate} \\ &= \bigvee_n \mathbf{F}^{n+1}(\perp)\end{aligned}$$

Teorema Knaster-Tarski pentru CPO

Demonstrație

Fie (C, \leq) o mulțime parțial ordonată completă și $\mathbf{F} : C \rightarrow C$ o funcție continuă.

- Arătăm că $a = \bigvee_n \mathbf{F}^n(\perp)$ este punct fix, i.e. $\mathbf{F}(a) = a$

$$\begin{aligned}\mathbf{F}(a) &= \mathbf{F}(\bigvee_n \mathbf{F}^n(\perp)) \\ &= \bigvee_n \mathbf{F}(\mathbf{F}^n(\perp)) \text{ din continuitate} \\ &= \bigvee_n \mathbf{F}^{n+1}(\perp) \\ &= \bigvee_n \mathbf{F}^n(\perp) = a\end{aligned}$$

Teorema Knaster-Tarski pentru CPO

Demonstrație (cont.)

- Arătăm că a este cel mai mic punct fix.

Teorema Knaster-Tarski pentru CPO

Demonstrație (cont.)

- Arătăm că a este cel mai mic punct fix.

Fie b un alt punct fix, i.e. $\mathbf{F}(b) = b$.

Demonstrăm prin inducție după $n \geq 1$ că $\mathbf{F}^n(\perp) \leq b$.

Teorema Knaster-Tarski pentru CPO

Demonstrație (cont.)

- Arătăm că a este cel mai mic punct fix.

Fie b un alt punct fix, i.e. $\mathbf{F}(b) = b$.

Demonstrăm prin inducție după $n \geq 1$ că $\mathbf{F}^n(\perp) \leq b$.

Pentru $n = 0$, $\mathbf{F}^0(\perp) = \perp \leq b$ deoarece \perp este prim element.

Teorema Knaster-Tarski pentru CPO

Demonstrație (cont.)

- Arătăm că a este cel mai mic punct fix.

Fie b un alt punct fix, i.e. $\mathbf{F}(b) = b$.

Demonstrăm prin inducție după $n \geq 1$ că $\mathbf{F}^n(\perp) \leq b$.

Pentru $n = 0$, $\mathbf{F}^0(\perp) = \perp \leq b$ deoarece \perp este prim element.

Dacă $\mathbf{F}^n(\perp) \leq b$, atunci $\mathbf{F}^{n+1}(\perp) \leq \mathbf{F}(b)$, deoarece \mathbf{F} este crescătoare.

Deoarece $\mathbf{F}(b) = b$ rezultă $\mathbf{F}^{n+1}(\perp) \leq b$.

Teorema Knaster-Tarski pentru CPO

Demonstrație (cont.)

- Arătăm că a este cel mai mic punct fix.

Fie b un alt punct fix, i.e. $\mathbf{F}(b) = b$.

Demonstrăm prin inducție după $n \geq 1$ că $\mathbf{F}^n(\perp) \leq b$.

Pentru $n = 0$, $\mathbf{F}^0(\perp) = \perp \leq b$ deoarece \perp este prim element.

Dacă $\mathbf{F}^n(\perp) \leq b$, atunci $\mathbf{F}^{n+1}(\perp) \leq \mathbf{F}(b)$, deoarece \mathbf{F} este crescătoare.

Deoarece $\mathbf{F}(b) = b$ rezultă $\mathbf{F}^{n+1}(\perp) \leq b$.

Stim $\mathbf{F}^n(\perp) \leq b$ oricare $n \geq 1$, deci $a = \bigvee_n \mathbf{F}^n(\perp) \leq b$.

Teorema Knaster-Tarski pentru CPO

Demonstrație (cont.)

- Arătăm că a este cel mai mic punct fix.

Fie b un alt punct fix, i.e. $\mathbf{F}(b) = b$.

Demonstrăm prin inducție după $n \geq 1$ că $\mathbf{F}^n(\perp) \leq b$.

Pentru $n = 0$, $\mathbf{F}^0(\perp) = \perp \leq b$ deoarece \perp este prim element.

Dacă $\mathbf{F}^n(\perp) \leq b$, atunci $\mathbf{F}^{n+1}(\perp) \leq \mathbf{F}(b)$, deoarece \mathbf{F} este crescătoare.

Deoarece $\mathbf{F}(b) = b$ rezultă $\mathbf{F}^{n+1}(\perp) \leq b$.

Stim $\mathbf{F}^n(\perp) \leq b$ oricare $n \geq 1$, deci $a = \bigvee_n \mathbf{F}^n(\perp) \leq b$.

Am arătat că a este cel mai mic punct fix al funcției \mathbf{F} . □

Completitudinea sistemului de deducție CDP

Clauze definite și funcții monotone

Fie A mulțimea variabilelor propozitionale (atomilor) p_1, p_2, \dots care apar în \mathcal{S} .

Clauze definite și funcții monotone

Fie At mulțimea variabilelor propozitionale (atomilor) p_1, p_2, \dots care apar în \mathcal{S} .

Fie $Baza = \{p_i \mid p_i \in \mathcal{S}\}$ mulțimea faptelor din \mathcal{S} .

Clauze definite și funcții monotone

Fie At mulțimea variabilelor propozitionale (atomilor) p_1, p_2, \dots care apar în \mathcal{S} .

Fie $Baza = \{p_i \mid p_i \in \mathcal{S}\}$ mulțimea faptelor din \mathcal{S} .

Exemplu

oslo	→	windy
oslo	→	norway
norway	→	cold
cold \wedge windy	→	winterIsComing
		oslo

$At = \{oslo, windy, norway, cold, winterIsComing\}$

$Baza = \{oslo\}$

Clauze definite și funcții monotone

Fie At mulțimea atomilor p_1, p_2, \dots care apar în \mathcal{S} .

Fie $Baza = \{p_i \mid p_i \in \mathcal{S}\}$ mulțimea atomilor care apar în *faptele* din \mathcal{S} .

Definim funcția $f_{\mathcal{S}} : \mathcal{P}(At) \rightarrow \mathcal{P}(At)$ prin

$$f_{\mathcal{S}}(Y) = Y \cup Baza$$

$$\begin{aligned} & \cup \{a \in At \mid (s_1 \wedge \dots \wedge s_n \rightarrow a) \text{ este în } \mathcal{S}, \\ & \quad s_1 \in Y, \dots, s_n \in Y\} \end{aligned}$$

Clauze definite și funcții monotone

Fie At mulțimea atomilor p_1, p_2, \dots care apar în \mathcal{S} .

Fie $Baza = \{p_i \mid p_i \in \mathcal{S}\}$ mulțimea atomilor care apar în *faptele* din \mathcal{S} .

Definim funcția $f_{\mathcal{S}} : \mathcal{P}(At) \rightarrow \mathcal{P}(At)$ prin

$$\begin{aligned} f_{\mathcal{S}}(Y) = Y \cup Baza \\ \cup \{a \in At \mid (s_1 \wedge \dots \wedge s_n \rightarrow a) \text{ este în } \mathcal{S}, \\ s_1 \in Y, \dots, s_n \in Y\} \end{aligned}$$

Exercițiu. Arătați că funcția $f_{\mathcal{S}}$ este monotonă.

Clauze definite și funcții monotone

Fie At mulțimea atomilor p_1, p_2, \dots care apar în S .

Propoziție

Funcția $f_S : \mathcal{P}(At) \rightarrow \mathcal{P}(At)$ este continuă.

Clauze definite și funcții monotone

Fie At mulțimea atomilor p_1, p_2, \dots care apar în S .

Propoziție

Funcția $f_S : \mathcal{P}(At) \rightarrow \mathcal{P}(At)$ este continuă.

Demonstrație

Arătăm că dacă $Y_1 \subseteq Y_2 \subseteq Y_3 \subseteq \dots$ atunci $f_S(\bigcup_k Y_k) = \bigcup_k f_S(Y_k)$.

Clauze definite și funcții monotone

Fie At mulțimea atomilor p_1, p_2, \dots care apar în S .

Propoziție

Funcția $f_S : \mathcal{P}(At) \rightarrow \mathcal{P}(At)$ este continuă.

Demonstrație

Arătăm că dacă $Y_1 \subseteq Y_2 \subseteq Y_3 \subseteq \dots$ atunci $f_S(\bigcup_k Y_k) = \bigcup_k f_S(Y_k)$.

Din faptul că f_S este crescătoare rezultă $f_S(\bigcup_k Y_k) \supseteq \bigcup_k f_S(Y_k)$

Clauze definite și funcții monotone

Fie At mulțimea atomilor p_1, p_2, \dots care apar în S .

Propoziție

Funcția $f_S : \mathcal{P}(At) \rightarrow \mathcal{P}(At)$ este continuă.

Demonstrație

Arătăm că dacă $Y_1 \subseteq Y_2 \subseteq Y_3 \subseteq \dots$ atunci $f_S(\bigcup_k Y_k) = \bigcup_k f_S(Y_k)$.

Din faptul că f_S este crescătoare rezultă $f_S(\bigcup_k Y_k) \supseteq \bigcup_k f_S(Y_k)$

Demonstrăm în continuare că $f_S(\bigcup_k Y_k) \subseteq \bigcup_k f_S(Y_k)$. Fie $a \in f_S(\bigcup_n Y_k)$. Sunt posibile trei cazuri

Clauze definite și funcții monotone

Fie At mulțimea atomilor p_1, p_2, \dots care apar în S .

Propoziție

Funcția $f_S : \mathcal{P}(At) \rightarrow \mathcal{P}(At)$ este continuă.

Demonstrație

Arătăm că dacă $Y_1 \subseteq Y_2 \subseteq Y_3 \subseteq \dots$ atunci $f_S(\bigcup_k Y_k) = \bigcup_k f_S(Y_k)$.

Din faptul că f_S este crescătoare rezultă $f_S(\bigcup_k Y_k) \supseteq \bigcup_k f_S(Y_k)$

Demonstrăm în continuare că $f_S(\bigcup_k Y_k) \subseteq \bigcup_k f_S(Y_k)$. Fie $a \in f_S(\bigcup_n Y_k)$. Sunt posibile trei cazuri

$a \in \bigcup_k Y_k$

Există un $k \geq 1$ astfel încât $a \in Y_k$, deci $a \in f_S(Y_k) \subseteq \bigcup_k f_S(Y_k)$.

Clauze definite și funcții monotone

Fie At mulțimea atomilor p_1, p_2, \dots care apar în S .

Propoziție

Funcția $f_S : \mathcal{P}(At) \rightarrow \mathcal{P}(At)$ este continuă.

Demonstrație

Arătăm că dacă $Y_1 \subseteq Y_2 \subseteq Y_3 \subseteq \dots$ atunci $f_S(\bigcup_k Y_k) = \bigcup_k f_S(Y_k)$.

Din faptul că f_S este crescătoare rezultă $f_S(\bigcup_k Y_k) \supseteq \bigcup_k f_S(Y_k)$

Demonstrăm în continuare că $f_S(\bigcup_k Y_k) \subseteq \bigcup_k f_S(Y_k)$. Fie $a \in f_S(\bigcup_n Y_k)$. Sunt posibile trei cazuri

$a \in \bigcup_k Y_k$

Există un $k \geq 1$ astfel încât $a \in Y_k$, deci $a \in f_S(Y_k) \subseteq \bigcup_k f_S(Y_k)$.

$a \in Baza \subseteq \bigcup_k f_S(Y_k)$

Clauze definite și funcții monotone

Fie At mulțimea atomilor p_1, p_2, \dots care apar în S .

Propoziție

Funcția $f_S : \mathcal{P}(At) \rightarrow \mathcal{P}(At)$ este continuă.

Demonstrație

Arătăm că dacă $Y_1 \subseteq Y_2 \subseteq Y_3 \subseteq \dots$ atunci $f_S(\bigcup_k Y_k) = \bigcup_k f_S(Y_k)$.

Din faptul că f_S este crescătoare rezultă $f_S(\bigcup_k Y_k) \supseteq \bigcup_k f_S(Y_k)$.

Demonstrăm în continuare că $f_S(\bigcup_k Y_k) \subseteq \bigcup_k f_S(Y_k)$. Fie $a \in f_S(\bigcup_n Y_k)$. Sunt posibile trei cazuri

- $a \in \bigcup_k Y_k$

Există un $k \geq 1$ astfel încât $a \in Y_k$, deci $a \in f_S(Y_k) \subseteq \bigcup_k f_S(Y_k)$.

- $a \in Baza \subseteq \bigcup_k f_S(Y_k)$

- Există s_1, \dots, s_n în $\bigcup_k Y_k$ astfel încât $(s_1 \wedge \dots \wedge s_n \rightarrow a)$ este în S .

Clauze definite și funcții monotone

Demonstrație (cont.)

- Există s_1, \dots, s_n în $\bigcup_k Y_k$ astfel încât $(s_1 \wedge \dots \wedge s_n \rightarrow a)$ este în \mathcal{S} .

Clauze definite și funcții monotone

Demonstrație (cont.)

□ Există s_1, \dots, s_n în $\bigcup_k Y_k$ astfel încât $(s_1 \wedge \dots \wedge s_n \rightarrow a)$ este în \mathcal{S} .

Pentru fiecare $i \in \{1, \dots, n\}$ există $k_i \in \mathbb{N}$ astfel încât $s_i \in Y_{k_i}$.

Clauze definite și funcții monotone

Demonstrație (cont.)

□ Există s_1, \dots, s_n în $\bigcup_k Y_k$ astfel încât $(s_1 \wedge \dots \wedge s_n \rightarrow a)$ este în \mathcal{S} .

Pentru fiecare $i \in \{1, \dots, n\}$ există $k_i \in \mathbb{N}$ astfel încât $s_i \in Y_{k_i}$.

Dacă $k_0 = \max\{k_1, \dots, k_n\}$ atunci $Y_{k_i} \subseteq Y_{k_0}$ pentru orice $i \in \{1, \dots, n\}$.

Clauze definite și funcții monotone

Demonstrație (cont.)

□ Există s_1, \dots, s_n în $\bigcup_k Y_k$ astfel încât $(s_1 \wedge \dots \wedge s_n \rightarrow a)$ este în \mathcal{S} .

Pentru fiecare $i \in \{1, \dots, n\}$ există $k_i \in \mathbb{N}$ astfel încât $s_i \in Y_{k_i}$.

Dacă $k_0 = \max\{k_1, \dots, k_n\}$ atunci $Y_{k_i} \subseteq Y_{k_0}$ pentru orice $i \in \{1, \dots, n\}$.

Rezultă că $s_1, \dots, s_n \in Y_{k_0}$, deci $a \in f_{\mathcal{S}}(Y_{k_0}) \subseteq \bigcup_k f_{\mathcal{S}}(Y_k)$.

Clauze definite și funcții monotone

Demonstrație (cont.)

□ Există s_1, \dots, s_n în $\bigcup_k Y_k$ astfel încât $(s_1 \wedge \dots \wedge s_n \rightarrow a)$ este în \mathcal{S} .

Pentru fiecare $i \in \{1, \dots, n\}$ există $k_i \in \mathbb{N}$ astfel încât $s_i \in Y_{k_i}$.

Dacă $k_0 = \max\{k_1, \dots, k_n\}$ atunci $Y_{k_i} \subseteq Y_{k_0}$ pentru orice $i \in \{1, \dots, n\}$.

Rezultă că $s_1, \dots, s_n \in Y_{k_0}$, deci $a \in f_{\mathcal{S}}(Y_{k_0}) \subseteq \bigcup_k f_{\mathcal{S}}(Y_k)$.

Am demonstrat că $f_{\mathcal{S}}$ este continuă.

□

Clauze definite și funcții monotone

Pentru funcția continuă $f_S : \mathcal{P}(At) \rightarrow \mathcal{P}(At)$

$$f_S(Y) = Y \cup \text{Baza}$$

$\cup \{a \in At \mid (s_1 \wedge \dots \wedge s_n \rightarrow a) \text{ este în } S,$

$$s_1 \in Y, \dots, s_n \in Y\}$$

aplicând [Teorema Knaster-Tarski pentru CPO](#), obținem că

$$\bigcup_n f_S^n(\emptyset)$$

este cel mai mic punct fix al lui f_S .

Clauze definite și funcții monotone

- Analizați ce se întamplă când considerăm succesiv

$$\emptyset, \quad f_S(\emptyset), \quad f_S(f_S(\emptyset)), \quad f_S(f_S(f_S(\emptyset))), \dots$$

La fiecare aplicare a lui f_S , rezultatul fie se mărește, fie rămâne neschimbat.

Clauze definite și funcții monotone

- Analizați ce se întamplă când considerăm succesiv

$$\emptyset, \quad f_S(\emptyset), \quad f_S(f_S(\emptyset)), \quad f_S(f_S(f_S(\emptyset))), \dots$$

La fiecare aplicare a lui f_S , rezultatul fie se mărește, fie rămâne neschimbat.

- Să presupunem că în S avem k atomi. Atunci după $k + 1$ aplicări ale lui f_S , trebuie să existe un punct în sirul de mulțimi obținute de unde o nouă aplicare a lui f_S nu mai schimbă rezultatul (**punct fix**):

$$f_S(X) = X$$

Clauze definite și funcții monotone

- Analizați ce se întamplă când considerăm succesiv

$$\emptyset, \quad f_S(\emptyset), \quad f_S(f_S(\emptyset)), \quad f_S(f_S(f_S(\emptyset))), \dots$$

La fiecare aplicare a lui f_S , rezultatul fie se mărește, fie rămâne neschimbăt.

- Să presupunem că în S avem k atomi. Atunci după $k + 1$ aplicări ale lui f_S , trebuie să existe un punct în sirul de mulțimi obținute de unde o nouă aplicare a lui f_S nu mai schimbă rezultatul (**punct fix**):

$$f_S(X) = X$$

- Dacă aplicăm f_S succesiv ca mai devreme până găsim un X cu proprietatea $f_S(X) = X$, atunci găsim **cel mai mic punct fix** al lui f_S .

Cel mai mic punct fix

Exemplu

$\begin{array}{ccc} cold & \rightarrow & wet \\ wet \wedge cold & \rightarrow & scotland \end{array}$

$$f_S(Y) = Y \cup \text{Baza}$$

$\cup \{a \in At \mid (s_1 \wedge \dots \wedge s_n \rightarrow a) \text{ este în } S,$
 $s_1 \in Y, \dots, s_n \in Y\}$

Se observă că $f_S(\emptyset) =$

Cel mai mic punct fix

Exemplu

$\begin{array}{ccc} cold & \rightarrow & wet \\ wet \wedge cold & \rightarrow & scotland \end{array}$

$$f_S(Y) = Y \cup \text{Baza}$$

$\cup \{a \in At \mid (s_1 \wedge \dots \wedge s_n \rightarrow a) \text{ este în } S,$
 $s_1 \in Y, \dots, s_n \in Y\}$

Se observă că $f_S(\emptyset) = \emptyset$, deci \emptyset este cel mai mic punct fix.

De aici deducem că niciun atom nu este consecință logică a formulelor de mai sus.

Exemplu

Exemplu

<i>cold</i>		
<i>cold</i>	→	<i>wet</i>
<i>windy</i>	→	<i>dry</i>
<i>wet</i> \wedge <i>cold</i>	→	<i>scotland</i>

$$f_S(Y) = Y \cup \text{Baza}$$

$\cup \{a \in At \mid (s_1 \wedge \dots \wedge s_n \rightarrow a) \text{ este în } S,$
 $s_1 \in Y, \dots, s_n \in Y\}$

Exemplu

Exemplu

<i>cold</i>		
<i>cold</i>	→	<i>wet</i>
<i>windy</i>	→	<i>dry</i>
<i>wet</i> \wedge <i>cold</i>	→	<i>scotland</i>

$$f_S(Y) = Y \cup \text{Baza}$$

$\cup \{a \in At \mid (s_1 \wedge \dots \wedge s_n \rightarrow a) \text{ este în } S,$
 $s_1 \in Y, \dots, s_n \in Y\}$

$$f_S(\emptyset) = \{ \text{cold} \}$$

Exemplu

Exemplu

<i>cold</i>	
<i>cold</i>	→ <i>wet</i>
<i>windy</i>	→ <i>dry</i>
<i>wet</i> \wedge <i>cold</i>	→ <i>scotland</i>

$$f_S(Y) = Y \cup \text{Baza}$$

$\cup \{ a \in At \mid (s_1 \wedge \dots \wedge s_n \rightarrow a) \text{ este în } S,$
 $s_1 \in Y, \dots, s_n \in Y \}$

$$f_S(\emptyset) = \{ \text{cold} \}$$

$$f_S(\{ \text{cold} \}) = \{ \text{cold}, \text{wet} \}$$

Exemplu

Exemplu

<i>cold</i>		
<i>cold</i>	→	<i>wet</i>
<i>windy</i>	→	<i>dry</i>
<i>wet</i> \wedge <i>cold</i>	→	<i>scotland</i>

$$f_S(Y) = Y \cup \text{Baza}$$

$\cup \{ a \in At \mid (s_1 \wedge \dots \wedge s_n \rightarrow a) \text{ este în } S,$
 $s_1 \in Y, \dots, s_n \in Y \}$

$$f_S(\emptyset) = \{ \text{cold} \}$$

$$f_S(\{ \text{cold} \}) = \{ \text{cold}, \text{wet} \}$$

$$f_S(\{ \text{cold}, \text{wet} \}) = \{ \text{cold}, \text{wet}, \text{scotland} \}$$

Exemplu

Exemplu

<i>cold</i>		
<i>cold</i>	→	<i>wet</i>
<i>windy</i>	→	<i>dry</i>
<i>wet</i> \wedge <i>cold</i>	→	<i>scotland</i>

$$f_S(Y) = Y \cup \text{Baza}$$

$\cup \{ a \in At \mid (s_1 \wedge \dots \wedge s_n \rightarrow a) \text{ este în } S,$
 $s_1 \in Y, \dots, s_n \in Y \}$

$$f_S(\emptyset) = \{ \text{cold} \}$$

$$f_S(\{ \text{cold} \}) = \{ \text{cold}, \text{wet} \}$$

$$f_S(\{ \text{cold}, \text{wet} \}) = \{ \text{cold}, \text{wet}, \text{scotland} \}$$

$$f_S(\{ \text{cold}, \text{wet}, \text{scotland} \}) = \{ \text{cold}, \text{wet}, \text{scotland} \}$$

Exemplu

Exemplu

<i>cold</i>		
<i>cold</i>	→	<i>wet</i>
<i>windy</i>	→	<i>dry</i>
<i>wet</i> \wedge <i>cold</i>	→	<i>scotland</i>

$$f_S(Y) = Y \cup \text{Baza}$$

$\cup \{ a \in At \mid (s_1 \wedge \dots \wedge s_n \rightarrow a) \text{ este în } S,$
 $s_1 \in Y, \dots, s_n \in Y \}$

$$f_S(\emptyset) = \{ \text{cold} \}$$

$$f_S(\{ \text{cold} \}) = \{ \text{cold}, \text{wet} \}$$

$$f_S(\{ \text{cold}, \text{wet} \}) = \{ \text{cold}, \text{wet}, \text{scotland} \}$$

$$f_S(\{ \text{cold}, \text{wet}, \text{scotland} \}) = \{ \text{cold}, \text{wet}, \text{scotland} \}$$

Deci cel mai mic punct fix este $\{ \text{cold}, \text{wet}, \text{scotland} \}$.

Programe logice și cel mai mic punct fix

Teoremă

Fie X este cel mai mic punct fix al funcției f_S . Atunci

$$q \in X \quad \text{dacă} \quad S \models q.$$

Intuiție: Cel mai mic punct fix al funcției f_S este mulțimea tuturor atomilor care sunt consecințe logice ale programului.

Funcția $f_S : \mathcal{P}(At) \rightarrow \mathcal{P}(At)$ este definită prin

$$\begin{aligned} f_S(Y) = Y \cup \text{Baza} \\ \cup \{a \in At \mid (s_1 \wedge \dots \wedge s_n \rightarrow a) \text{ este în } S, s_1 \in Y, \dots, s_n \in Y\} \end{aligned}$$

unde At este mulțimea atomilor din S și $\text{Baza} = \{p_i \mid p_i \in S\}$ este mulțimea atomilor care apar în faptele din S .

Programe logice și cel mai mic punct fix

Demonstrație

$(\Rightarrow) q \in X \Rightarrow \mathcal{S} \models q.$

- Funcția $f_{\mathcal{S}}$ conservă atomii adevărați.
- Deci, dacă fiecare clauză unitate din \mathcal{S} este adevărată, după fiecare aplicare a funcției $f_{\mathcal{S}}$ obținem o mulțime adevărată de atomi.

Programe logice și cel mai mic punct fix

Demonstrație

$(\Rightarrow) q \in X \Rightarrow \mathcal{S} \models q.$

- Funcția $f_{\mathcal{S}}$ conservă atomii adevărați.
- Deci, dacă fiecare clauză unitate din \mathcal{S} este adevărată, după fiecare aplicare a funcției $f_{\mathcal{S}}$ obținem o mulțime adevărată de atomi.

$(\Leftarrow) \mathcal{S} \models q \Rightarrow q \in X.$

- Fie $\mathcal{S} \models q$. Presupunem prin absurd că $q \notin X$.
- Căutăm o evaluare e care face fiecare clauză din \mathcal{S} adevărată, dar q falsă.

Programe logice și cel mai mic punct fix

Demonstrație (cont.)

□ Fie evaluarea

$$e(p) = \begin{cases} 1, & \text{dacă } p \in X \\ 0, & \text{altfel} \end{cases}$$

Programe logice și cel mai mic punct fix

Demonstrație (cont.)

- Fie evaluarea

$$e(p) = \begin{cases} 1, & \text{dacă } p \in X \\ 0, & \text{altfel} \end{cases}$$

- Evident, această interpretare face q falsă.

Programe logice și cel mai mic punct fix

Demonstrație (cont.)

- Fie evaluarea

$$e(p) = \begin{cases} 1, & \text{dacă } p \in X \\ 0, & \text{altfel} \end{cases}$$

- Evident, această interpretare face q falsă.
- Arătăm că $e^+(P) = 1$, pentru orice clauză $P \in \mathcal{S}$.

Programe logice și cel mai mic punct fix

Demonstrație (cont.)

- Fie evaluarea

$$e(p) = \begin{cases} 1, & \text{dacă } p \in X \\ 0, & \text{altfel} \end{cases}$$

- Evident, această interpretare face q falsă.
- Arătăm că $e^+(P) = 1$, pentru orice clauză $P \in \mathcal{S}$.
- Fie $P \in \mathcal{S}$. Avem două cazuri:

Programe logice și cel mai mic punct fix

Demonstrație (cont.)

- Fie evaluarea

$$e(p) = \begin{cases} 1, & \text{dacă } p \in X \\ 0, & \text{altfel} \end{cases}$$

- Evident, această interpretare face q falsă.
- Arătăm că $e^+(P) = 1$, pentru orice clauză $P \in \mathcal{S}$.
- Fie $P \in \mathcal{S}$. Avem două cazuri:
 - 1 P este un fapt. Atunci $P \in X$, deci $e(P) = 1$.

Programe logice și cel mai mic punct fix

Demonstrație (cont.)

- Fie evaluarea

$$e(p) = \begin{cases} 1, & \text{dacă } p \in X \\ 0, & \text{altfel} \end{cases}$$

- Evident, această interpretare face q falsă.
- Arătăm că $e^+(P) = 1$, pentru orice clauză $P \in \mathcal{S}$.
- Fie $P \in \mathcal{S}$. Avem două cazuri:
 - 1 P este un fapt. Atunci $P \in X$, deci $e(P) = 1$.
 - 2 P este de forma $p_1 \wedge \dots \wedge p_n \rightarrow r$. Atunci avem două cazuri:

Programe logice și cel mai mic punct fix

Demonstrație (cont.)

- Fie evaluarea

$$e(p) = \begin{cases} 1, & \text{dacă } p \in X \\ 0, & \text{altfel} \end{cases}$$

- Evident, această interpretare face q falsă.
- Arătăm că $e^+(P) = 1$, pentru orice clauză $P \in \mathcal{S}$.
- Fie $P \in \mathcal{S}$. Avem două cazuri:
 - 1 P este un fapt. Atunci $P \in X$, deci $e(P) = 1$.
 - 2 P este de forma $p_1 \wedge \dots \wedge p_n \rightarrow r$. Atunci avem două cazuri:
 - există un p_i , $i = 1, \dots, n$, care nu este în X . Deci $e^+(P) = 1$.

Programe logice și cel mai mic punct fix

Demonstrație (cont.)

- Fie evaluarea

$$e(p) = \begin{cases} 1, & \text{dacă } p \in X \\ 0, & \text{altfel} \end{cases}$$

- Evident, această interpretare face q falsă.
- Arătăm că $e^+(P) = 1$, pentru orice clauză $P \in \mathcal{S}$.
- Fie $P \in \mathcal{S}$. Avem două cazuri:

1 P este un fapt. Atunci $P \in X$, deci $e(P) = 1$.

2 P este de forma $p_1 \wedge \dots \wedge p_n \rightarrow r$. Atunci avem două cazuri:

- există un p_i , $i = 1, \dots, n$, care nu este în X . Deci $e^+(P) = 1$.
- toți p_i , $i = 1, \dots, n$, sunt în X . Atunci $r \in f_{\mathcal{S}}(X) = X$, deci $e(r) = 1$. În concluzie $e^+(P) = 1$.



Sistemul de deducție

Corolar

Sistemul de deducție pentru clauze definite propoziționale este complet pentru a arăta clauze unitate:

dacă $\mathcal{S} \models q$, atunci $\mathcal{S} \vdash q$.

Sistemul de deducție

Corolar

Sistemul de deducție pentru clauze definite propoziționale este complet pentru a arăta clauze unitate:

dacă $\mathcal{S} \models q$, atunci $\mathcal{S} \vdash q$.

Demonstrație

- Presupunem $\mathcal{S} \models q$.
- Atunci $q \in X$, unde X este cel mai mic punct fix al funcției $f_{\mathcal{S}}$.
- Fiecare aplicare a funcției $f_{\mathcal{S}}$ produce o mulțime demonstrabilă de atomi.
- Cum cel mai mic punct fix este atins după un număr finit de aplicări ale lui $f_{\mathcal{S}}$, orice $a \in X$ are o derivare.

□

Bibliografie

- J.W. Lloyd, Foundations of Logic Programming, Second Edition, Springer, 1987
- R.J. Brachman, H.J. Levesque, Knowledge Representation and Reasoning, Morgan Kaufmann Publishers, San Francisco, CA, 2004
- Logic Programming, The University of Edinburgh,
<https://www.inf.ed.ac.uk/teaching/courses/lp/>



Pe săptămâna viitoare!

Curs 7

2021-2022

Fundamentele Limbajelor de Programare

Cuprins

- 1 Rezoluție SLD - cazul logicii propoziționale**
- 2 Logica de ordinul I - recapitulare**
- 3 Logica Horn**

Rezoluție SLD - cazul logicii propoziționale

Metodă de decizie

Avem o **metodă de decizie** (*decision procedure*) pentru a verifica $S \vdash q$

Metoda constă în:

- calcularea celui mai mic punct fix X al funcției f_S
- dacă $q \in X$ atunci returnăm **true**, altfel returnăm **false**

Metodă de decizie

Avem o **metodă de decizie** (*decision procedure*) pentru a verifica $S \vdash q$

Metoda constă în:

- calcularea celui mai mic punct fix X al funcției f_S
- dacă $q \in X$ atunci returnăm **true**, altfel returnăm **false**

Această metodă se termină.

Exercițiu. De ce?

Metodă de decizie

Avem o **metodă de decizie** (*decision procedure*) pentru a verifica $S \vdash q$

Metoda constă în:

- calcularea celui mai mic punct fix X al funcției f_S
- dacă $q \in X$ atunci returnăm **true**, altfel returnăm **false**

Această metodă se termină.

Exercițiu. De ce?

Program Prolog = baza de cunoștințe

- Un program Prolog reprezintă o bază de cunoștințe (knowledge base) KB. Cel mai mic punct fix al funcției f_{KB} definește totalitatea cunoștiințelor care pot fi deduse din KB.

Metodă de decizie

Avem o **metodă de decizie** (*decision procedure*) pentru a verifica $S \vdash q$

Metoda constă în:

- calcularea celui mai mic punct fix X al funcției f_S
- dacă $q \in X$ atunci returnăm **true**, altfel returnăm **false**

Această metodă se termină.

Exercițiu. De ce?

Program Prolog = baza de cunoștințe

- Un program Prolog reprezintă o bază de cunoștințe (knowledge base) KB. Cel mai mic punct fix al funcției f_{KB} definește totalitatea cunoștiințelor care pot fi deduse din KB.
- Pentru o bază de cunoștințe formată numai din clauze propoziționale definite, cel mai mic punct fix poate fi calculat în timp liniar.

Clauze definite

- Singurele formule admise sunt de forma:
 - q
 - $p_1 \wedge \dots \wedge p_n \rightarrow q$, unde toate p_i, q sunt variabile propozitionale.
- O clauză definită $p_1 \wedge \dots \wedge p_n \rightarrow q$ poate fi gândită ca formula
 $\neg p_1 \vee \dots \vee \neg p_n \vee q$

Clauze definite

- Singurele formule admise sunt de forma:
 - q
 - $p_1 \wedge \dots \wedge p_n \rightarrow q$, unde toate p_i, q sunt variabile propozitionale.
- O clauză definită $p_1 \wedge \dots \wedge p_n \rightarrow q$ poate fi gândită ca formula
 $\neg p_1 \vee \dots \vee \neg p_n \vee q$

Echivalent, putem reprezenta clauza definită de mai sus și prin
 $\{\neg p_1, \dots, \neg p_n, q\}$

Calculul celui mai mic punct fix

KB:

$\{oslo\}$

$\{\neg oslo, windy\}$

$\{\neg oslo, norway\}$

$\{\neg norway, cold\}$

$\{\neg cold, \neg windy, winter\}$

LFP:

Calculul celui mai mic punct fix

LFP:

$\{\neg \text{oslo}, \text{windy}\}$

$\{\text{oslo}\}$

$\{\neg \text{oslo}, \text{nорway}\}$

$\{\neg \text{nорway}, \text{cold}\}$

$\{\neg \text{cold}, \neg \text{windy}, \text{winter}\}$

Calculul celui mai mic punct fix

LFP:

$\{\neg \text{oslo}, \text{windy}\}$

$\{\text{oslo}\}$

$\{\neg \text{oslo}, \text{nорway}\}$

$\{\neg \text{nорway}, \text{cold}\}$

$\{\neg \text{cold}, \neg \text{windy}, \text{winter}\}$

Calculul celui mai mic punct fix

LFP:

$\{\neg \text{norway}, \text{cold}\}$

$\{\text{oslo}\}$

$\{\neg \text{cold}, \neg \text{windy}, \text{winter}\}$

$\{\text{windy}\}$

$\{\text{norway}\}$

Calculul celui mai mic punct fix

LFP:

$\{\neg \text{norway}, \text{cold}\}$

$\{\text{oslo}\}$

$\{\neg \text{cold}, \neg \text{windy}, \text{winter}\}$

$\{\text{windy}\}$

$\{\text{norway}\}$

Calculul celui mai mic punct fix

LFP:

$\{\neg cold, \neg windy, winter\}$

$\{oslo\}$

$\{windy\}$

$\{norway\}$

$\{cold\}$

Calculul celui mai mic punct fix

LFP:

$\{\neg cold, \neg windy, winter\}$

$\{oslo\}$

$\{windy\}$

$\{norway\}$

$\{cold\}$

Calculul celui mai mic punct fix

LFP:

$\{\neg \text{windy}, \text{winter}\}$

$\{\text{oslo}\}$

$\{\text{windy}\}$

$\{\text{norway}\}$

$\{\text{cold}\}$

Calculul celui mai mic punct fix

LFP:

$\{\neg \text{windy}, \text{winter}\}$

$\{\text{oslo}\}$

$\{\text{windy}\}$

$\{\text{norway}\}$

$\{\text{cold}\}$

Calculul celui mai mic punct fix

LFP:

{*oslo*}

{*windy*}

{*norway*}

{*cold*}

{**winter**}

Propagarea unității

- În procedeul anterior am folosit o metodă asemănătoare rezoluției în care una din clauze are un singur literal.
- Clauzele formate dintr-un singur literal se numesc **clauze unitate** (*unit clause*), iar metoda anterioară se numește **propagarea unității** (*unit propagation*).
- Printr-o reprezentare adecvată a datelor, propagarea unității poate fi implementată în timp liniar în raport cu dimensiunea bazei de cunoștințe inițiale.
- **Clauzele Horn propoziționale** sunt clauze care au cel mult un literal pozitiv. Clauzele propoziționale definite sunt clauze Horn care au exact un literal pozitiv. Folosind metoda de propagare a unității problema satisfiabilității pentru clauze Horn propoziționale HORNSAT poate fi rezolvată în timp liniar.

Forward chaining / Backward chaining

- Metoda anterioară este centrată pe *lărgirea bazei de cunoștințe*.
- Pentru a afla răspunsul la o întrebare (-? winter) adăugăm pas cu pas cunoștințe noi, verificând de fiecare dată dacă am răspuns la întrebare.
- Acest procedeu se numește **forward chaining**.

Forward chaining / Backward chaining

- Metoda anterioară este centrată pe *lărgirea bazei de cunoștințe*.
- Pentru a afla răspunsul la o întrebare (-? winter) adăugăm pas cu pas cunoștințe noi, verificând de fiecare dată dacă am răspuns la întrebare.
- Acest procedeu se numește **forward chaining**.

Nu acesta este algoritmul folosit de Prolog!

Forward chaining / Backward chaining

- Metoda anterioară este centrată pe *lărgirea bazei de cunoștințe*.
- Pentru a afla răspunsul la o întrebare (-? winter) adăugăm pas cu pas cunoștințe noi, verificând de fiecare dată dacă am răspuns la întrebare.
- Acest procedeu se numește **forward chaining**.

Nu acesta este algoritmul folosit de Prolog!

- Metoda folosită de Prolog se numește **backward chaining**. Această metodă este centrată pe *găsirea răspunsului la întrebare*.

Backward chaining

- În *backward chaining* pornim de la întrebare (`-? winter`) și analizăm baza de cunoștinte, căutând o regulă care are drept concluzie scopul (`winter :- cold, windy`).
- În continuare vom încerca să satisfacem scopurile noi (`cold` și `windy`) prin același procedeu.
- Această metodă este realizată printr-o implementare particulară a rezoluției - rezoluția SLD.

Rezoluția SLD (cazul propozițional)

Fie S o mulțime de clauze definite.

SLD

$$\frac{\neg p_1 \vee \cdots \vee \neg q \vee \cdots \vee \neg p_n}{\neg p_1 \vee \cdots \vee \neg q_1 \vee \cdots \vee \neg q_m \vee \cdots \vee \neg p_n}$$

unde $q \vee \neg q_1 \vee \cdots \vee \neg q_m$ este o clauză definită din S .

Rezoluția SLD

Fie S o mulțime de clauze definite și q o întrebare.

O derivare din S prin rezoluție SLD este o secvență

$$G_0 := \neg q, \quad G_1, \quad \dots, \quad G_k, \quad \dots$$

în care G_{i+1} se obține din G_i prin regula **SLD**.

Dacă există un k cu $G_k = \square$ (clauza vidă), atunci derivarea se numește **SLD-respingere**.

Rezoluția SLD

Teoremă (Completitudinea SLD-rezoluției)

Sunt echivalente:

- există o *SLD-respingere* a lui q din S ,
- $S \vdash q$,
- $S \models q$.

Rezoluția SLD

Baza de cunoștințe KB:

```
oslo .  
windy :- oslo.  
norway :- oslo.  
cold :- norway.  
winter :- cold, windy.
```

Întrebarea:

```
-? winter.
```

Rezoluția SLD

Baza de cunoștințe KB:

```
oslo .  
windy :- oslo.  
norway :- oslo.  
cold :- norway.  
winter :- cold, windy.
```

Întrebarea:

-? winter.

- Formă clauzală:

$$KB = \{\{oslo\}, \{\neg oslo, windy\}, \{\neg oslo, norway\}, \{\neg norway, cold\}, \\ \{\neg cold, \neg windy, winter\}\}$$

- $KB \vdash winter$ dacă și numai dacă $KB \cup \{\neg winter\}$ este satisfiabilă.

Rezoluția SLD

Baza de cunoștințe KB:

```
oslo .  
windy :- oslo.  
norway :- oslo.  
cold :- norway.  
winter :- cold, windy.
```

Întrebarea:

-? winter.

- Formă clauzală:

$$KB = \{\{oslo\}, \{\neg oslo, windy\}, \{\neg oslo, norway\}, \{\neg norway, cold\}, \\ \{\neg cold, \neg windy, winter\}\}$$

- $KB \vdash winter$ dacă și numai dacă $KB \cup \{\neg winter\}$ este satisfiabilă.
- Satisfiabilitatea este verificată prin rezoluție

SLD = Linear resolution with Selected literal for Definite clauses

Clause Horn propoziționale - rezoluția SLD

Exemplu

Demonstrăm $\text{KB} \vdash \text{winter}$ prin rezoluție SLD:

$\{\neg\text{winter}\}$

Clause Horn propoziționale - rezoluția SLD

Exemplu

Demonstrăm $\text{KB} \vdash \text{winter}$ prin rezoluție SLD:

$\{\neg\text{winter}\}$

$\{\neg\text{cold}, \neg\text{windy}, \text{winter}\}$

$\{\neg\text{cold}, \neg\text{windy}\}$

Clause Horn propoziționale - rezoluția SLD

Exemplu

Demonstrăm $\text{KB} \vdash \text{winter}$ prin rezoluție SLD:

$$\{\neg\text{winter}\} \qquad \qquad \{\neg\text{cold}, \neg\text{windy}, \text{winter}\}$$

$$\{\neg\text{cold}, \neg\text{windy}\} \qquad \{\neg\text{norway}, \text{cold}\}$$

$$\{\neg\text{norway}, \neg\text{windy}\}$$

Clause Horn propoziționale - rezoluția SLD

Exemplu

Demonstrăm $\text{KB} \vdash \text{winter}$ prin rezoluție SLD:

$$\{\neg\text{winter}\} \qquad \{\neg\text{cold}, \neg\text{windy}, \text{winter}\}$$

$$\{\neg\text{cold}, \neg\text{windy}\} \qquad \{\neg\text{norway}, \text{cold}\}$$

$$\{\neg\text{norway}, \neg\text{windy}\} \qquad \{\neg\text{oslo}, \text{norway}\}$$

$$\{\neg\text{oslo}, \neg\text{windy}\}$$

Clause Horn propoziționale - rezoluția SLD

Exemplu

Demonstrăm $\text{KB} \vdash \text{winter}$ prin rezoluție SLD:

$$\{\neg\text{winter}\} \qquad \{\neg\text{cold}, \neg\text{windy}, \text{winter}\}$$

$$\{\neg\text{cold}, \neg\text{windy}\} \qquad \{\neg\text{norway}, \text{cold}\}$$

$$\{\neg\text{norway}, \neg\text{windy}\} \qquad \{\neg\text{oslo}, \text{norway}\}$$

$$\{\neg\text{oslo}, \neg\text{windy}\} \qquad \{\text{oslo}\}$$

$$\{\neg\text{windy}\}$$

Clause Horn propoziționale - rezoluția SLD

Exemplu

Demonstrăm KB $\vdash \text{winter}$ prin rezoluție SLD:

$$\{\neg\text{winter}\} \qquad \{\neg\text{cold}, \neg\text{windy}, \text{winter}\}$$

$$\{\neg\text{cold}, \neg\text{windy}\} \qquad \{\neg\text{norway}, \text{cold}\}$$

$$\{\neg\text{norway}, \neg\text{windy}\} \qquad \{\neg\text{oslo}, \text{norway}\}$$

$$\{\neg\text{oslo}, \neg\text{windy}\} \qquad \{\text{oslo}\}$$

$$\{\neg\text{windy}\} \qquad \{\neg\text{oslo}, \text{windy}\}$$

$$\{\neg\text{oslo}\}$$

Clause Horn propoziționale - rezoluția SLD

Exemplu

Demonstrăm $\text{KB} \vdash \text{winter}$ prin rezoluție SLD:

$$\{\neg\text{winter}\} \qquad \{\neg\text{cold}, \neg\text{windy}, \text{winter}\}$$

$$\{\neg\text{cold}, \neg\text{windy}\} \qquad \{\neg\text{norway}, \text{cold}\}$$

$$\{\neg\text{norway}, \neg\text{windy}\} \qquad \{\neg\text{oslo}, \text{norway}\}$$

$$\{\neg\text{oslo}, \neg\text{windy}\} \qquad \{\text{oslo}\}$$

$$\{\neg\text{windy}\} \qquad \{\neg\text{oslo}, \text{windy}\}$$

$$\{\neg\text{oslo}\} \qquad \{\text{oslo}\}$$

□

Clause Horn propoziționale - rezoluția SLD

Exemplu

Demonstrăm $\text{KB} \vdash \text{winter}$ prin rezoluție SLD:

$$\{\neg\text{winter}\} \qquad \{\neg\text{cold}, \neg\text{windy}, \text{winter}\}$$

$$\{\neg\text{cold}, \neg\text{windy}\} \qquad \{\neg\text{norway}, \text{cold}\}$$

$$\{\neg\text{norway}, \neg\text{windy}\} \qquad \{\neg\text{oslo}, \text{norway}\}$$

$$\{\neg\text{oslo}, \neg\text{windy}\} \qquad \{\text{oslo}\}$$

$$\{\neg\text{windy}\} \qquad \{\neg\text{oslo}, \text{windy}\}$$

$$\{\neg\text{oslo}\} \qquad \{\text{oslo}\}$$

□

În continuare vom studia aceste mecanisme în logica de ordinul I.

Logica de ordinul I - recapitulare

Limbaje de ordinul I

Un **limbaj \mathcal{L} de ordinul I** este format din:

- o mulțime numărabilă de **variabile** $V = \{x_n \mid n \in \mathbb{N}\}$
- conectorii** $\neg, \rightarrow, \wedge, \vee$
- paranteze
- cuantificatorul universal** \forall și **cuantificatorul existențial** \exists
- o mulțime **R** de **simboluri de relații**
- o mulțime **F** de **simboluri de funcții**
- o mulțime **C** de **simboluri de constante**
- o funcție **aritate ar** : $F \cup R \rightarrow \mathbb{N}^*$

Logica de ordinul I

- \mathcal{L} este unic determinat de $\tau = (\mathbf{R}, \mathbf{F}, \mathbf{C}, \text{ari})$
- τ se numește **signatura** (vocabularul, alfabetul) lui \mathcal{L}

Logica de ordinul I

- \mathcal{L} este unic determinat de $\tau = (\mathbf{R}, \mathbf{F}, \mathbf{C}, \text{ari})$
- τ se numește **signatura** (vocabularul, alfabetul) lui \mathcal{L}

Exemplu

Un limbaj \mathcal{L} de ordinul I în care:

- $\mathbf{R} = \{P, R\}$
- $\mathbf{F} = \{f\}$
- $\mathbf{C} = \{c\}$
- $\text{ari}(P) = 1, \text{ari}(R) = 2, \text{ari}(f) = 2$

Sintaxa Prolog

Atenție!

- În sintaxa Prolog
 - termenii compuși sunt predicate: `father(eddard, jon_snow)`
 - operatorii sunt funcții: `+`, `*`, `mod`
- Sintaxa Prolog nu face diferență între **simboluri de funcții** și **simboluri de predicate**!
- Dar este important când ne uităm la teoria corespunzătoare programului în logică să facem acestă distincție.

Logica de ordinul I

Termenii lui \mathcal{L} sunt definiți inductiv astfel:

- orice variabilă este un termen;
- orice simbol de constantă este un termen;
- dacă $f \in \mathbf{F}$, $ar(f) = n$ și t_1, \dots, t_n sunt termeni, atunci $f(t_1, \dots, t_n)$ este termen.

Notăm cu $Trm_{\mathcal{L}}$ mulțimea termenilor lui \mathcal{L} .

Logica de ordinul I

Termenii lui \mathcal{L} sunt definiți inductiv astfel:

- orice variabilă este un termen;
- orice simbol de constantă este un termen;
- dacă $f \in \mathbf{F}$, $ar(f) = n$ și t_1, \dots, t_n sunt termeni, atunci $f(t_1, \dots, t_n)$ este termen.

Notăm cu $Trm_{\mathcal{L}}$ mulțimea termenilor lui \mathcal{L} .

Exemplu

$$c, \quad x_1, \quad f(x_1, c), \quad f(f(x_2, x_2), c)$$

Logica de ordinul I

Formulele atomice ale lui \mathcal{L} sunt definite astfel:

- dacă $R \in \mathbf{R}$, $ar(R) = n$ și t_1, \dots, t_n sunt termeni, atunci $R(t_1, \dots, t_n)$ este formulă atomică.

Logica de ordinul I

Formulele atomice ale lui \mathcal{L} sunt definite astfel:

- dacă $R \in \mathbf{R}$, $ar(R) = n$ și t_1, \dots, t_n sunt termeni, atunci $R(t_1, \dots, t_n)$ este formulă atomică.

Exemplu

$$P(f(x_1, c)), \quad R(c, x_3)$$

Logica de ordinul I

Formulele lui \mathcal{L} sunt definite astfel:

- orice formulă atomică este o formulă
- dacă φ este o formulă, atunci $\neg\varphi$ este o formulă
- dacă φ și ψ sunt formule, atunci $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \rightarrow \psi$ sunt formule
- dacă φ este o formulă și x este o variabilă, atunci $\forall x \varphi$, $\exists x \varphi$ sunt formule

Logica de ordinul I

Formulele lui \mathcal{L} sunt definite astfel:

- orice formulă atomică este o formulă
- dacă φ este o formulă, atunci $\neg\varphi$ este o formulă
- dacă φ și ψ sunt formule, atunci $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \rightarrow \psi$ sunt formule
- dacă φ este o formulă și x este o variabilă, atunci $\forall x \varphi$, $\exists x \varphi$ sunt formule

Exemplu

$$P(f(x_1, c)), \quad P(x_1) \vee P(c), \quad \forall x_1 P(x_1), \quad \forall x_2 R(x_2, x_1)$$

Logica de ordinul I

Exemplu

Fie limbajul \mathcal{L}_1 cu $\mathbf{R} = \{<\}$, $\mathbf{F} = \{s, +\}$, $\mathbf{C} = \{0\}$ și
 $ari(s) = 1$, $ari(+)$ = $ari(<)$ = 2.

Logica de ordinul I

Exemplu

Fie limbajul \mathcal{L}_1 cu $\mathbf{R} = \{<\}$, $\mathbf{F} = \{s, +\}$, $\mathbf{C} = \{0\}$ și
 $ari(s) = 1$, $ari(+) = ari(<) = 2$.

Exemple de **termeni**:

Logica de ordinul I

Exemplu

Fie limbajul \mathcal{L}_1 cu $\mathbf{R} = \{<\}$, $\mathbf{F} = \{s, +\}$, $\mathbf{C} = \{0\}$ și
 $ari(s) = 1$, $ari(+)$ = $ari(<)$ = 2.

Exemple de **termeni**:

0, x , $s(0)$, $s(s(0))$, $s(x)$, $s(s(x))$, ...,

Logica de ordinul I

Exemplu

Fie limbajul \mathcal{L}_1 cu $\mathbf{R} = \{<\}$, $\mathbf{F} = \{s, +\}$, $\mathbf{C} = \{0\}$ și
 $ari(s) = 1$, $ari(+)$ = $ari(<)$ = 2.

Exemple de **termeni**:

$0, x, s(0), s(s(0)), s(x), s(s(x)), \dots,$
 $+ (0, 0), +(s(s(0))), +(0, s(0))), +(x, s(0)), +(x, s(x)), \dots,$

Logica de ordinul I

Exemplu

Fie limbajul \mathcal{L}_1 cu $\mathbf{R} = \{\langle\}$, $\mathbf{F} = \{s, +\}$, $\mathbf{C} = \{0\}$ și
 $ari(s) = 1$, $ari(+)$ = $ari(\langle) = 2$.

Exemple de **termeni**:

$0, x, s(0), s(s(0)), s(x), s(s(x)), \dots,$
 $+(0, 0), +(s(s(0))), +(0, s(0))), +(x, s(0)), +(x, s(x)), \dots,$

Exemple de **formule atomice**:

$\langle (0, 0), \langle (x, 0), \langle (s(s(x)), s(0)), \dots$

Logica de ordinul I

Exemplu

Fie limbajul \mathcal{L}_1 cu $\mathbf{R} = \{<\}$, $\mathbf{F} = \{s, +\}$, $\mathbf{C} = \{0\}$ și $ari(s) = 1$, $ari(+)$ = $ari(<)$ = 2.

Exemplu de **termeni**:

$0, x, s(0), s(s(0)), s(x), s(s(x)), \dots,$
 $+ (0, 0), +(s(s(0))), +(0, s(0))), +(x, s(0)), +(x, s(x)), \dots,$

Exemplu de **formule atomice**:

$< (0, 0), < (x, 0), < (s(s(x)), s(0)), \dots$

Exemplu de **formule**:

$\forall x \forall y < (x, +(x, y))$
 $\forall x < (x, s(x))$

Semantica

Pentru a stabili dacă o formulă este adevărată, avem nevoie de o interpretare într-o structură!

Structură

Definiție

O **structură** este de forma $\mathcal{A} = (A, \mathbf{F}^{\mathcal{A}}, \mathbf{R}^{\mathcal{A}}, \mathbf{C}^{\mathcal{A}})$, unde

- A este o mulțime nevidă
 - $\mathbf{F}^{\mathcal{A}} = \{f^{\mathcal{A}} \mid f \in \mathbf{F}\}$ este o mulțime de operații pe A ;
dacă f are aritatea n , atunci $f^{\mathcal{A}} : A^n \rightarrow A$.
 - $\mathbf{R}^{\mathcal{A}} = \{R^{\mathcal{A}} \mid R \in \mathbf{R}\}$ este o mulțime de relații pe A ;
dacă R are aritatea n , atunci $R^{\mathcal{A}} \subseteq A^n$.
 - $\mathbf{C}^{\mathcal{A}} = \{c^{\mathcal{A}} \in A \mid c \in \mathbf{C}\}$.
-
- A se numește **universul** structurii \mathcal{A} .
 - $f^{\mathcal{A}}$ (respectiv $R^{\mathcal{A}}, c^{\mathcal{A}}$) se numește **interpretarea** lui f (respectiv R, c) în \mathcal{A} .

Structură

Exemplu

$\mathcal{L}_1 : \mathbf{R} = \{<\}, \mathbf{F} = \{s, +\}, \mathbf{C} = \{0\}$ cu $ari(s) = 1$, $ari(+) = ari(<) = 2$.

$\mathcal{N} = (\mathbb{N}, s^{\mathcal{N}}, +^{\mathcal{N}}, <^{\mathcal{N}}, 0^{\mathcal{N}})$ unde

- $s^{\mathcal{N}} : \mathbb{N} \rightarrow \mathbb{N}$, $s^{\mathcal{N}}(n) := n + 1$,
- $+^{\mathcal{N}} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, $+^{\mathcal{N}}(n, m) := n + m$,
- $<^{\mathcal{N}} \subseteq \mathbb{N} \times \mathbb{N}$, $<^{\mathcal{N}} = \{(n, m) \mid n < m\}$,
- $0^{\mathcal{N}} := 0$

Interpretare

Fie \mathcal{L} un limbaj de ordinul I și \mathcal{A} o (\mathcal{L} -)structură.

Definiție

O **interpretare a variabilelor** lui \mathcal{L} în \mathcal{A} este o funcție

$$I : V \rightarrow A.$$

Interpretare

Fie \mathcal{L} un limbaj de ordinul I și \mathcal{A} o (\mathcal{L} -)structură.

Definiție

O **interpretare a variabilelor** lui \mathcal{L} în \mathcal{A} este o funcție

$$I : V \rightarrow A.$$

Definiție

Inductiv, definim **interpretarea termenului** t în \mathcal{A} sub I ($t_I^{\mathcal{A}}$) prin:

- dacă $t = x_i \in V$, atunci $t_I^{\mathcal{A}} := I(x_i)$
- dacă $t = c \in \mathbf{C}$, atunci $t_I^{\mathcal{A}} := c^{\mathcal{A}}$
- dacă $t = f(t_1, \dots, t_n)$, atunci $t_I^{\mathcal{A}} := f^{\mathcal{A}}((t_1)_I^{\mathcal{A}}, \dots, (t_n)_I^{\mathcal{A}})$

Interpretare

Definim inductiv faptul că o formulă este adevărată în \mathcal{A} sub interpretarea / astfel:

Interpretare

Definim inductiv faptul că o formulă este adevărată în \mathcal{A} sub interpretarea I astfel:

- $\mathcal{A}, I \models P(t_1, \dots, t_n)$ dacă $P^{\mathcal{A}}((t_1)_I^{\mathcal{A}}, \dots, (t_n)_I^{\mathcal{A}})$

Interpretare

Definim inductiv faptul că o formulă este adevărată în \mathcal{A} sub interpretarea I astfel:

- $\mathcal{A}, I \models P(t_1, \dots, t_n)$ dacă $P^{\mathcal{A}}((t_1)_I^{\mathcal{A}}, \dots, (t_n)_I^{\mathcal{A}})$
- $\mathcal{A}, I \models \neg\varphi$ dacă $\mathcal{A}, I \not\models \varphi$

Interpretare

Definim inductiv faptul că o formulă este adevărată în \mathcal{A} sub interpretarea I astfel:

- $\mathcal{A}, I \models P(t_1, \dots, t_n)$ dacă $P^{\mathcal{A}}((t_1)_I^{\mathcal{A}}, \dots, (t_n)_I^{\mathcal{A}})$
- $\mathcal{A}, I \models \neg\varphi$ dacă $\mathcal{A}, I \not\models \varphi$
- $\mathcal{A}, I \models \varphi \vee \psi$ dacă $\mathcal{A}, I \models \varphi$ sau $\mathcal{A}, I \models \psi$

Interpretare

Definim inductiv faptul că o formulă este adevărată în \mathcal{A} sub interpretarea I astfel:

- $\mathcal{A}, I \models P(t_1, \dots, t_n)$ dacă $P^{\mathcal{A}}((t_1)_I^{\mathcal{A}}, \dots, (t_n)_I^{\mathcal{A}})$
- $\mathcal{A}, I \models \neg\varphi$ dacă $\mathcal{A}, I \not\models \varphi$
- $\mathcal{A}, I \models \varphi \vee \psi$ dacă $\mathcal{A}, I \models \varphi$ sau $\mathcal{A}, I \models \psi$
- $\mathcal{A}, I \models \varphi \wedge \psi$ dacă $\mathcal{A}, I \models \varphi$ și $\mathcal{A}, I \models \psi$

Interpretare

Definim inductiv faptul că o formulă este adevărată în \mathcal{A} sub interpretarea I astfel:

- $\mathcal{A}, I \models P(t_1, \dots, t_n)$ dacă $P^{\mathcal{A}}((t_1)_I^{\mathcal{A}}, \dots, (t_n)_I^{\mathcal{A}})$
- $\mathcal{A}, I \models \neg\varphi$ dacă $\mathcal{A}, I \not\models \varphi$
- $\mathcal{A}, I \models \varphi \vee \psi$ dacă $\mathcal{A}, I \models \varphi$ sau $\mathcal{A}, I \models \psi$
- $\mathcal{A}, I \models \varphi \wedge \psi$ dacă $\mathcal{A}, I \models \varphi$ și $\mathcal{A}, I \models \psi$
- $\mathcal{A}, I \models \varphi \rightarrow \psi$ dacă $\mathcal{A}, I \not\models \varphi$ sau $\mathcal{A}, I \models \psi$

Interpretare

Definim inductiv faptul că o formulă este adevărată în \mathcal{A} sub interpretarea I astfel:

- $\mathcal{A}, I \models P(t_1, \dots, t_n)$ dacă $P^{\mathcal{A}}((t_1)_I^{\mathcal{A}}, \dots, (t_n)_I^{\mathcal{A}})$
- $\mathcal{A}, I \models \neg\varphi$ dacă $\mathcal{A}, I \not\models \varphi$
- $\mathcal{A}, I \models \varphi \vee \psi$ dacă $\mathcal{A}, I \models \varphi$ sau $\mathcal{A}, I \models \psi$
- $\mathcal{A}, I \models \varphi \wedge \psi$ dacă $\mathcal{A}, I \models \varphi$ și $\mathcal{A}, I \models \psi$
- $\mathcal{A}, I \models \varphi \rightarrow \psi$ dacă $\mathcal{A}, I \not\models \varphi$ sau $\mathcal{A}, I \models \psi$
- $\mathcal{A}, I \models \forall x \varphi$ dacă pentru orice $a \in A$ avem $\mathcal{A}, I_{x_i \leftarrow a} \models \varphi$
- $\mathcal{A}, I \models \exists x \varphi$ dacă există $a \in A$ astfel încât $\mathcal{A}, I_{x_i \leftarrow a} \models \varphi$

unde pentru orice $a \in A$, $I_{x \leftarrow a}(y) = \begin{cases} I(y) & \text{dacă } y \neq x \\ a & \text{dacă } y = x \end{cases}$

Interpretare

- O formulă φ este **adevărată într-o structură** \mathcal{A} , notat $\mathcal{A} \models \varphi$, dacă este adevărată în \mathcal{A} sub orice interpretare.
Spunem că \mathcal{A} este **model** al lui φ .
- O formulă φ este **adevărată în logica de ordinul I**, notat $\models \varphi$, dacă este adevărată în orice structură.

Model

Exemplu

Fie limbajul \mathcal{L} cu $\mathbf{F} = \{s\}$, $\mathbf{R} = \{P\}$, $\mathbf{C} = \{0\}$ cu $ari(s) = ari(P) = 1$.

Model

Exemplu

Fie limbajul \mathcal{L} cu $\mathbf{F} = \{s\}$, $\mathbf{R} = \{P\}$, $\mathbf{C} = \{0\}$ cu $ari(s) = ari(P) = 1$.

Fie structura $\mathcal{N} = (\mathbb{N}, s^{\mathcal{N}}, P^{\mathcal{N}}, 0^{\mathcal{N}})$ unde $0^{\mathcal{N}} := 1$ și

- $s^{\mathcal{N}} : \mathbb{N} \rightarrow \mathbb{N}$, $s^{\mathcal{N}}(n) := n^2$
- $P^{\mathcal{N}} \subset \mathbb{N}$, $P^{\mathcal{N}} = \{n \mid n \text{ este impar}\}$

Model

Exemplu

Fie limbajul \mathcal{L} cu $\mathbf{F} = \{s\}$, $\mathbf{R} = \{P\}$, $\mathbf{C} = \{0\}$ cu $ari(s) = ari(P) = 1$.

Fie structura $\mathcal{N} = (\mathbb{N}, s^{\mathcal{N}}, P^{\mathcal{N}}, 0^{\mathcal{N}})$ unde $0^{\mathcal{N}} := 1$ și

- $s^{\mathcal{N}} : \mathbb{N} \rightarrow \mathbb{N}$, $s^{\mathcal{N}}(n) := n^2$
- $P^{\mathcal{N}} \subset \mathbb{N}$, $P^{\mathcal{N}} = \{n \mid n \text{ este impar}\}$

Demonstrați că $\mathcal{N} \models \forall x (P(x) \rightarrow P(s(x)))$.

Model

Exemplu

Fie limbajul \mathcal{L} cu $\mathbf{F} = \{s\}$, $\mathbf{R} = \{P\}$, $\mathbf{C} = \{0\}$ cu $ari(s) = ari(P) = 1$.

Fie structura $\mathcal{N} = (\mathbb{N}, s^{\mathcal{N}}, P^{\mathcal{N}}, 0^{\mathcal{N}})$ unde $0^{\mathcal{N}} := 1$ și

- $s^{\mathcal{N}} : \mathbb{N} \rightarrow \mathbb{N}$, $s^{\mathcal{N}}(n) := n^2$
- $P^{\mathcal{N}} \subset \mathbb{N}$, $P^{\mathcal{N}} = \{n \mid n \text{ este impar}\}$

Demonstrați că $\mathcal{N} \models \forall x (P(x) \rightarrow P(s(x)))$.

Fie $I : V \rightarrow \mathbb{N}$ o interpretare. Observăm că $\mathcal{N}, I \models P(x)$ dacă $P^{\mathcal{N}}(I(x))$, adică

Model

Exemplu

Fie limbajul \mathcal{L} cu $\mathbf{F} = \{s\}$, $\mathbf{R} = \{P\}$, $\mathbf{C} = \{0\}$ cu $ari(s) = ari(P) = 1$.

Fie structura $\mathcal{N} = (\mathbb{N}, s^{\mathcal{N}}, P^{\mathcal{N}}, 0^{\mathcal{N}})$ unde $0^{\mathcal{N}} := 1$ și

- $s^{\mathcal{N}} : \mathbb{N} \rightarrow \mathbb{N}$, $s^{\mathcal{N}}(n) := n^2$
- $P^{\mathcal{N}} \subset \mathbb{N}$, $P^{\mathcal{N}} = \{n \mid n \text{ este impar}\}$

Demonstrați că $\mathcal{N} \models \forall x (P(x) \rightarrow P(s(x)))$.

Fie $I : V \rightarrow \mathbb{N}$ o interpretare. Observăm că

$\mathcal{N}, I \models P(x)$ dacă $P^{\mathcal{N}}(I(x))$, adică $\mathcal{N}, I \models P(x)$ dacă $I(x)$ este impar.

Model

Exemplu

Fie limbajul \mathcal{L} cu $\mathbf{F} = \{s\}$, $\mathbf{R} = \{P\}$, $\mathbf{C} = \{0\}$ cu $ari(s) = ari(P) = 1$.

Fie structura $\mathcal{N} = (\mathbb{N}, s^{\mathcal{N}}, P^{\mathcal{N}}, 0^{\mathcal{N}})$ unde $0^{\mathcal{N}} := 1$ și

- $s^{\mathcal{N}} : \mathbb{N} \rightarrow \mathbb{N}$, $s^{\mathcal{N}}(n) := n^2$
- $P^{\mathcal{N}} \subset \mathbb{N}$, $P^{\mathcal{N}} = \{n \mid n \text{ este impar}\}$

Demonstrați că $\mathcal{N} \models \forall x (P(x) \rightarrow P(s(x)))$.

Fie $I : V \rightarrow \mathbb{N}$ o interpretare. Observăm că

$\mathcal{N}, I \models P(x)$ dacă $P^{\mathcal{N}}(I(x))$, adică $\mathcal{N}, I \models P(x)$ dacă $I(x)$ este impar.

$\mathcal{N}, I \models \forall x (P(x) \rightarrow P(s(x)))$ dacă

Model

Exemplu

Fie limbajul \mathcal{L} cu $\mathbf{F} = \{s\}$, $\mathbf{R} = \{P\}$, $\mathbf{C} = \{0\}$ cu $ari(s) = ari(P) = 1$.

Fie structura $\mathcal{N} = (\mathbb{N}, s^{\mathcal{N}}, P^{\mathcal{N}}, 0^{\mathcal{N}})$ unde $0^{\mathcal{N}} := 1$ și

- $s^{\mathcal{N}} : \mathbb{N} \rightarrow \mathbb{N}$, $s^{\mathcal{N}}(n) := n^2$
- $P^{\mathcal{N}} \subset \mathbb{N}$, $P^{\mathcal{N}} = \{n \mid n \text{ este impar}\}$

Demonstrați că $\mathcal{N} \models \forall x (P(x) \rightarrow P(s(x)))$.

Fie $I : V \rightarrow \mathbb{N}$ o interpretare. Observăm că

$\mathcal{N}, I \models P(x)$ dacă $P^{\mathcal{N}}(I(x))$, adică $\mathcal{N}, I \models P(x)$ dacă $I(x)$ este impar.

$\mathcal{N}, I \models \forall x (P(x) \rightarrow P(s(x)))$ dacă

$\mathcal{N}, I_{x \leftarrow n} \models P(x) \rightarrow P(s(x))$ oricare $n \in N$

Model

Exemplu

Fie limbajul \mathcal{L} cu $\mathbf{F} = \{s\}$, $\mathbf{R} = \{P\}$, $\mathbf{C} = \{0\}$ cu $ari(s) = ari(P) = 1$.

Fie structura $\mathcal{N} = (\mathbb{N}, s^{\mathcal{N}}, P^{\mathcal{N}}, 0^{\mathcal{N}})$ unde $0^{\mathcal{N}} := 1$ și

- $s^{\mathcal{N}} : \mathbb{N} \rightarrow \mathbb{N}$, $s^{\mathcal{N}}(n) := n^2$
- $P^{\mathcal{N}} \subset \mathbb{N}$, $P^{\mathcal{N}} = \{n \mid n \text{ este impar}\}$

Demonstrați că $\mathcal{N} \models \forall x (P(x) \rightarrow P(s(x)))$.

Fie $I : V \rightarrow \mathbb{N}$ o interpretare. Observăm că

$\mathcal{N}, I \models P(x)$ dacă $P^{\mathcal{N}}(I(x))$, adică $\mathcal{N}, I \models P(x)$ dacă $I(x)$ este impar.

$\mathcal{N}, I \models \forall x (P(x) \rightarrow P(s(x)))$ dacă

$\mathcal{N}, I_{x \leftarrow n} \models P(x) \rightarrow P(s(x))$ oricare $n \in N$

$\mathcal{N}, I_{x \leftarrow n} \not\models P(x)$ sau $\mathcal{N}, I_{x \leftarrow n} \models P(s(x))$ oricare $n \in N$

Model

Exemplu

Fie limbajul \mathcal{L} cu $\mathbf{F} = \{s\}$, $\mathbf{R} = \{P\}$, $\mathbf{C} = \{0\}$ cu $ari(s) = ari(P) = 1$.

Fie structura $\mathcal{N} = (\mathbb{N}, s^{\mathcal{N}}, P^{\mathcal{N}}, 0^{\mathcal{N}})$ unde $0^{\mathcal{N}} := 1$ și

- $s^{\mathcal{N}} : \mathbb{N} \rightarrow \mathbb{N}$, $s^{\mathcal{N}}(n) := n^2$
- $P^{\mathcal{N}} \subset \mathbb{N}$, $P^{\mathcal{N}} = \{n \mid n \text{ este impar}\}$

Demonstrați că $\mathcal{N} \models \forall x (P(x) \rightarrow P(s(x)))$.

Fie $I : V \rightarrow \mathbb{N}$ o interpretare. Observăm că

$\mathcal{N}, I \models P(x)$ dacă $P^{\mathcal{N}}(I(x))$, adică $\mathcal{N}, I \models P(x)$ dacă $I(x)$ este impar.

$\mathcal{N}, I \models \forall x (P(x) \rightarrow P(s(x)))$ dacă

$\mathcal{N}, I_{x \leftarrow n} \models P(x) \rightarrow P(s(x))$ oricare $n \in N$

$\mathcal{N}, I_{x \leftarrow n} \not\models P(x)$ sau $\mathcal{N}, I_{x \leftarrow n} \models P(s(x))$ oricare $n \in N$

$I_{x \leftarrow n}(x)$ nu este impar sau $I_{x \leftarrow n}(s(x))$ este impar oricare $n \in \mathbb{N}$

Model

Exemplu

Fie limbajul \mathcal{L} cu $\mathbf{F} = \{s\}$, $\mathbf{R} = \{P\}$, $\mathbf{C} = \{0\}$ cu $ari(s) = ari(P) = 1$.

Fie structura $\mathcal{N} = (\mathbb{N}, s^{\mathcal{N}}, P^{\mathcal{N}}, 0^{\mathcal{N}})$ unde $0^{\mathcal{N}} := 1$ și

- $s^{\mathcal{N}} : \mathbb{N} \rightarrow \mathbb{N}$, $s^{\mathcal{N}}(n) := n^2$
- $P^{\mathcal{N}} \subset \mathbb{N}$, $P^{\mathcal{N}} = \{n \mid n \text{ este impar}\}$

Demonstrați că $\mathcal{N} \models \forall x (P(x) \rightarrow P(s(x)))$.

Fie $I : V \rightarrow \mathbb{N}$ o interpretare. Observăm că

$\mathcal{N}, I \models P(x)$ dacă $P^{\mathcal{N}}(I(x))$, adică $\mathcal{N}, I \models P(x)$ dacă $I(x)$ este impar.

$\mathcal{N}, I \models \forall x (P(x) \rightarrow P(s(x)))$ dacă

$\mathcal{N}, I_{x \leftarrow n} \models P(x) \rightarrow P(s(x))$ oricare $n \in N$

$\mathcal{N}, I_{x \leftarrow n} \not\models P(x)$ sau $\mathcal{N}, I_{x \leftarrow n} \models P(s(x))$ oricare $n \in N$

$I_{x \leftarrow n}(x)$ nu este impar sau $I_{x \leftarrow n}(s(x))$ este impar oricare $n \in \mathbb{N}$

n este par sau n^2 este impar oricare $n \in \mathbb{N}$

Model

Exemplu

Fie limbajul \mathcal{L} cu $\mathbf{F} = \{s\}$, $\mathbf{R} = \{P\}$, $\mathbf{C} = \{0\}$ cu $ari(s) = ari(P) = 1$.

Fie structura $\mathcal{N} = (\mathbb{N}, s^{\mathcal{N}}, P^{\mathcal{N}}, 0^{\mathcal{N}})$ unde $0^{\mathcal{N}} := 1$ și

- $s^{\mathcal{N}} : \mathbb{N} \rightarrow \mathbb{N}$, $s^{\mathcal{N}}(n) := n^2$
- $P^{\mathcal{N}} \subset \mathbb{N}$, $P^{\mathcal{N}} = \{n \mid n \text{ este impar}\}$

Demonstrați că $\mathcal{N} \models \forall x (P(x) \rightarrow P(s(x)))$.

Fie $I : V \rightarrow \mathbb{N}$ o interpretare. Observăm că

$\mathcal{N}, I \models P(x)$ dacă $P^{\mathcal{N}}(I(x))$, adică $\mathcal{N}, I \models P(x)$ dacă $I(x)$ este impar.

$\mathcal{N}, I \models \forall x (P(x) \rightarrow P(s(x)))$ dacă

$\mathcal{N}, I_{x \leftarrow n} \models P(x) \rightarrow P(s(x))$ oricare $n \in N$

$\mathcal{N}, I_{x \leftarrow n} \not\models P(x)$ sau $\mathcal{N}, I_{x \leftarrow n} \models P(s(x))$ oricare $n \in N$

$I_{x \leftarrow n}(x)$ nu este impar sau $I_{x \leftarrow n}(s(x))$ este impar oricare $n \in \mathbb{N}$

n este par sau n^2 este impar oricare $n \in \mathbb{N}$

ceea ce este înțodeauna adevărat.

Consecință logică

Definiție

O formulă φ este o **consecință logică** a formulelor $\varphi_1, \dots, \varphi_n$, notată

$$\varphi_1, \dots, \varphi_n \models \varphi,$$

dacă pentru orice structură \mathcal{A}

dacă $\mathcal{A} \models \varphi_1$ și ... și $\mathcal{A} \models \varphi_n$, atunci $\mathcal{A} \models \varphi$

Consecință logică

Definiție

O formulă φ este o **consecință logică** a formulelor $\varphi_1, \dots, \varphi_n$, notată

$$\varphi_1, \dots, \varphi_n \models \varphi,$$

dacă pentru orice structură \mathcal{A}

dacă $\mathcal{A} \models \varphi_1$ și ... și $\mathcal{A} \models \varphi_n$, atunci $\mathcal{A} \models \varphi$

Problemă semidecidabilă!

Nu există algoritm care să decidă mereu dacă o formula este sau nu consecință logică a altei formule în logica de ordinul I!

Logica de ordinul I - sintaxa

Limbaj de ordinul I \mathcal{L}

- unic determinat de $\tau = (\mathbf{R}, \mathbf{F}, \mathbf{C}, \text{ari})$

Termenii lui \mathcal{L} , notați $\text{Trm}_{\mathcal{L}}$, sunt definiți inductiv astfel:

- orice variabilă este un termen;
- orice simbol de constantă este un termen;
- dacă $f \in \mathbf{F}$, $\text{ar}(f) = n$ și t_1, \dots, t_n sunt termeni, atunci $f(t_1, \dots, t_n)$ este termen.

Formulele atomice ale lui \mathcal{L} sunt definite astfel:

- dacă $R \in \mathbf{R}$, $\text{ar}(R) = n$ și t_1, \dots, t_n sunt termeni, atunci $R(t_1, \dots, t_n)$ este formulă atomică.

Formulele lui \mathcal{L} sunt definite astfel:

- orice formulă atomică este o formulă
- dacă φ este o formulă, atunci $\neg\varphi$ este o formulă
- dacă φ și ψ sunt formule, atunci $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \rightarrow \psi$ sunt formule
- dacă φ este o formulă și x este o variabilă, atunci $\forall x \varphi$, $\exists x \varphi$ sunt formule

Logica de ordinul I - semantică (optional)

O **structură** este de forma $\mathcal{A} = (A, \mathbf{F}^{\mathcal{A}}, \mathbf{R}^{\mathcal{A}}, \mathbf{C}^{\mathcal{A}})$, unde

- A este o mulțime nevidă
- $\mathbf{F}^{\mathcal{A}} = \{f^{\mathcal{A}} \mid f \in \mathbf{F}\}$ este o mulțime de operații pe A ; dacă f are aritatea n , atunci $f^{\mathcal{A}} : A^n \rightarrow A$.
- $\mathbf{R}^{\mathcal{A}} = \{R^{\mathcal{A}} \mid R \in \mathbf{R}\}$ este o mulțime de relații pe A ; dacă R are aritatea n , atunci $R^{\mathcal{A}} \subseteq A^n$.
- $\mathbf{C}^{\mathcal{A}} = \{c^{\mathcal{A}} \in A \mid c \in \mathbf{C}\}$.

O **interpretare a variabilelor** lui \mathcal{L} în \mathcal{A} (\mathcal{A} -interpretare) este o funcție $I : V \rightarrow A$.

Inductiv, definim **interpretarea termenului** t în \mathcal{A} sub I notat $t_I^{\mathcal{A}}$.

Inductiv, definim când o formulă este adevărată în \mathcal{A} în interpretarea I notat $\mathcal{A}, I \models \varphi$. În acest caz spunem că (\mathcal{A}, I) este **model** pentru φ .

O formulă φ este **adevărată într-o structură** \mathcal{A} , notat $\mathcal{A} \models \varphi$, dacă este adevărată în \mathcal{A} sub orice interpretare. Spunem că \mathcal{A} este **model** al lui φ .

O formulă φ este **adevărată în logica de ordinul I**, notat $\models \varphi$, dacă este adevărată în orice structură. O formulă φ este **validă** dacă $\models \varphi$.

O formulă φ este **satisfiabilă** dacă există o structură \mathcal{A} și o \mathcal{A} -interpretare I astfel încât $\mathcal{A}, I \models \varphi$.

Deducție și satisfiabilitate

Fie $\varphi_1, \dots, \varphi_n, \varphi$ formule în logica propozițională
(enunțuri în calculul cu predicate).

$\{\varphi_1, \dots, \varphi_n\} \models \varphi$ este echivalent cu

Deducție și satisfiabilitate

Fie $\varphi_1, \dots, \varphi_n, \varphi$ formule în logica propozițională
(enunțuri în calculul cu predicate).

$\{\varphi_1, \dots, \varphi_n\} \models \varphi$ este echivalent cu

$\models \varphi_1 \wedge \dots \wedge \varphi_n \rightarrow \varphi$ este echivalent cu

Deducție și satisfiabilitate

Fie $\varphi_1, \dots, \varphi_n, \varphi$ formule în logica propozițională
(enunțuri în calculul cu predicate).

$\{\varphi_1, \dots, \varphi_n\} \models \varphi$ este echivalent cu

$\models \varphi_1 \wedge \dots \wedge \varphi_n \rightarrow \varphi$ este echivalent cu

$\models \neg\varphi_1 \vee \dots \vee \neg\varphi_n \vee \varphi$ este echivalent cu

Deducție și satisfiabilitate

Fie $\varphi_1, \dots, \varphi_n, \varphi$ formule în logica propozițională
(enunțuri în calculul cu predicate).

$\{\varphi_1, \dots, \varphi_n\} \models \varphi$ este echivalent cu

$\models \varphi_1 \wedge \dots \wedge \varphi_n \rightarrow \varphi$ este echivalent cu

$\models \neg\varphi_1 \vee \dots \vee \neg\varphi_n \vee \varphi$ este echivalent cu

$\varphi_1 \wedge \dots \wedge \varphi_n \wedge \neg\varphi$ este satisfiabilă



Logica Horn

Literali

- În calculul propozițional un **literal** este o **variabilă** sau **negația unei variabile**.

literal := $p \mid \neg p$ unde p este variabilă propozițională

Literali

- În calculul propozițional un **literal** este o **variabilă** sau **negația unei variabile**.

literal := $p \mid \neg p$ unde p este variabilă propozițională

- În logica de ordinul I un **literal** este o **formulă atomică** sau **negația unei formule atomice**.

literal := $P(t_1, \dots, t_n) \mid \neg P(t_1, \dots, t_n)$

unde $P \in \mathbf{R}$, $\text{ari}(P) = n$, și t_1, \dots, t_n sunt termeni.

Clauze

- O clauză este o disjuncție de literali.

Clauze

- O clauză este o disjuncție de literali.
- Dacă L_1, \dots, L_n sunt literali atunci clauza $L_1 \vee \dots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \dots, L_n\}$

clauză = mulțime de literali

Clauze

- O clauză este o disjuncție de literali.
- Dacă L_1, \dots, L_n sunt literali atunci clauza $L_1 \vee \dots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \dots, L_n\}$
$$\text{clauză} = \text{mulțime de literali}$$
- Clauza $C = \{L_1, \dots, L_n\}$ este satisfiabilă dacă $L_1 \vee \dots \vee L_n$ este satisfiabilă.

Clauze

- O clauză este o disjuncție de literali.
- Dacă L_1, \dots, L_n sunt literali atunci clauza $L_1 \vee \dots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \dots, L_n\}$
$$\text{clauză} = \text{mulțime de literali}$$
- Clauza $C = \{L_1, \dots, L_n\}$ este satisfiabilă dacă $L_1 \vee \dots \vee L_n$ este satisfiabilă.
- O clauză C este trivială dacă conține un literal și complementul lui.

Clauze

- O clauză este o disjuncție de literali.
- Dacă L_1, \dots, L_n sunt literali atunci clauza $L_1 \vee \dots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \dots, L_n\}$
$$\text{clauză} = \text{mulțime de literali}$$
- Clauza $C = \{L_1, \dots, L_n\}$ este satisfiabilă dacă $L_1 \vee \dots \vee L_n$ este satisfiabilă.
- O clauză C este trivială dacă conține un literal și complementul lui.
- Când $n = 0$ obținem clauza vidă, care se notează \square

Clauze

- O clauză este o disjuncție de literali.
- Dacă L_1, \dots, L_n sunt literali atunci clauza $L_1 \vee \dots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \dots, L_n\}$
$$\text{clauză} = \text{mulțime de literali}$$
- Clauza $C = \{L_1, \dots, L_n\}$ este satisfiabilă dacă $L_1 \vee \dots \vee L_n$ este satisfiabilă.
- O clauză C este trivială dacă conține un literal și complementul lui.
- Când $n = 0$ obținem clauza vidă, care se notează \square
- Prin definiție, clauza \square nu este satisfiabilă.

Clauze

- O clauză este o disjuncție de literali.
- Dacă L_1, \dots, L_n sunt literali atunci clauza $L_1 \vee \dots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \dots, L_n\}$
$$\text{clauză} = \text{mulțime de literali}$$
- Clauza $C = \{L_1, \dots, L_n\}$ este satisfiabilă dacă $L_1 \vee \dots \vee L_n$ este satisfiabilă.
- O clauză C este trivială dacă conține un literal și complementul lui.
- Când $n = 0$ obținem clauza vidă, care se notează \square
- Prin definiție, clauza \square nu este satisfiabilă.

Rezoluția este o metodă de verificare a satisfiabilității unei mulțimi de clauze.

Clauze în logica de ordinul I

$$\{\neg Q_1, \dots, \neg Q_n, P_1, \dots, P_k\}$$

unde $n, k \geq 0$ și $Q_1, \dots, Q_n, P_1, \dots, P_k$ sunt formule atomice.

- formula corespunzătoare este

$$\forall x_1 \dots \forall x_m (\neg Q_1 \vee \dots \vee \neg Q_n \vee P_1 \vee \dots \vee P_k)$$

unde x_1, \dots, x_m sunt toate variabilele care apar în clauză

- echivalent, putem scrie

$$\forall x_1 \dots \forall x_m (Q_1 \wedge \dots \wedge Q_n \rightarrow P_1 \vee \dots \vee P_k)$$

- cuantificarea universală a clauzelor este implicită

$$Q_1 \wedge \dots \wedge Q_n \rightarrow P_1 \vee \dots \vee P_k$$

Clauze definite. Programe logice. Clauze Horn

□ clauză:

$$\{\neg Q_1, \dots, \neg Q_n, P_1, \dots, P_k\} \text{ sau } Q_1 \wedge \dots \wedge Q_n \rightarrow P_1 \vee \dots \vee P_k$$

unde $n, k \geq 0$ și $Q_1, \dots, Q_n, P_1, \dots, P_k$ sunt formule atomice.

Clauze definite. Programe logice. Clauze Horn

- clauză:

$$\{\neg Q_1, \dots, \neg Q_n, P_1, \dots, P_k\} \text{ sau } Q_1 \wedge \dots \wedge Q_n \rightarrow P_1 \vee \dots \vee P_k$$

unde $n, k \geq 0$ și $Q_1, \dots, Q_n, P_1, \dots, P_k$ sunt formule atomice.

- clauză program definită: $k = 1$

- cazul $n > 0$: $Q_1 \wedge \dots \wedge Q_n \rightarrow P$

- cazul $n = 0$: $\top \rightarrow P$ (clauză unitate, fapt)

Program logic definit = mulțime finită de clauze definite

Clauze definite. Programe logice. Clauze Horn

- clauză:

$$\{\neg Q_1, \dots, \neg Q_n, P_1, \dots, P_k\} \text{ sau } Q_1 \wedge \dots \wedge Q_n \rightarrow P_1 \vee \dots \vee P_k$$

unde $n, k \geq 0$ și $Q_1, \dots, Q_n, P_1, \dots, P_k$ sunt formule atomice.

- clauză program definită: $k = 1$

 - cazul $n > 0$: $Q_1 \wedge \dots \wedge Q_n \rightarrow P$

 - cazul $n = 0$: $\top \rightarrow P$ (clauză unitate, fapt)

Program logic definit = mulțime finită de clauze definite

- clauză scop definită (țintă, întrebare): $k=0$

 - $Q_1 \wedge \dots \wedge Q_n \rightarrow \perp$

- clauza vidă □: $n = k = 0$

Clauze definite. Programe logice. Clauze Horn

- clauză:

$$\{\neg Q_1, \dots, \neg Q_n, P_1, \dots, P_k\} \text{ sau } Q_1 \wedge \dots \wedge Q_n \rightarrow P_1 \vee \dots \vee P_k$$

unde $n, k \geq 0$ și $Q_1, \dots, Q_n, P_1, \dots, P_k$ sunt formule atomice.

- clauză program definită: $k = 1$

- cazul $n > 0$: $Q_1 \wedge \dots \wedge Q_n \rightarrow P$

- cazul $n = 0$: $\top \rightarrow P$ (clauză unitate, fapt)

Program logic definit = mulțime finită de clauze definite

- clauză scop definită (țintă, întrebare): $k=0$

- $Q_1 \wedge \dots \wedge Q_n \rightarrow \perp$

- clauza vidă □: $n = k = 0$

Clauza Horn = clauză program definită sau clauză scop ($k \leq 1$)

Clauze Horn țintă

□ clauză scop definită (țintă, întrebare): $Q_1 \wedge \dots \wedge Q_n \rightarrow \perp$

- fie x_1, \dots, x_m toate variabilele care apar în Q_1, \dots, Q_n
 $\forall x_1 \dots \forall x_m (\neg Q_1 \vee \dots \vee \neg Q_n) \models \neg \exists x_1 \dots \exists x_m (Q_1 \wedge \dots \wedge Q_n)$
- clauza țintă o vom scrie Q_1, \dots, Q_n

Negația unei "întrebări" în PROLOG este clauză Horn țintă.

Programare logica

- Logica clauzelor definite/Logica Horn: un fragment al logicii de ordinul I în care singurele formule admise sunt clauze Horn
 - formule atomice: $P(t_1, \dots, t_n)$
 - $Q_1 \wedge \dots \wedge Q_n \rightarrow P$
unde toate Q_i, P sunt formule atomice, \top sau \perp

Programare logica

- Logica clauzelor definite/Logica Horn: un fragment al logicii de ordinul I în care singurele formule admise sunt clauze Horn
 - formule atomice: $P(t_1, \dots, t_n)$
 - $Q_1 \wedge \dots \wedge Q_n \rightarrow P$
unde toate Q_i, P sunt formule atomice, \top sau \perp
- Problema programării logice: reprezentăm cunoștințele ca o mulțime de clauze definite KB și suntem interesați să aflăm răspunsul la o întrebare de forma $Q_1 \wedge \dots \wedge Q_n$, unde toate Q_i sunt formule atomice

$$KB \models Q_1 \wedge \dots \wedge Q_n$$

- Variabilele din KB sunt **cuantificate universal**.
- Variabilele din Q_1, \dots, Q_n sunt **cuantificate existențial**.

Limbajul PROLOG are la bază logica clauzelor Horn.

Logica clauzelor definite

Exemplu

Fie următoarele clauze definite:

$\text{father}(\text{jon}, \text{ken}).$

$\text{father}(\text{ken}, \text{liz}).$

$\text{father}(X, Y) \rightarrow \text{ancestor}(X, Y)$

$\text{daughter}(X, Y) \rightarrow \text{ancestor}(Y, X)$

$\text{ancestor}(X, Y) \wedge \text{ancestor}(Y, Z) \rightarrow \text{ancestor}(X, Z)$

Putem întreba:

- $\text{ancestor}(\text{jon}, \text{liz})$
- dacă există Q astfel încât $\text{ancestor}(Q, \text{ken})$
(adică $\exists Q \text{ancestor}(Q, \text{ken})$)



Pe săptămâna viitoare!

Curs 8

2021-2022

Fundamentele Limbajelor de Programare

Cuprins

- 1 Logica Horn**
- 2 Sistem de deducție pentru logica Horn**
- 3 Rezoluție SLD**



Logica Horn

Logica de ordinul I - sintaxa

Limbaj de ordinul I \mathcal{L}

- unic determinat de $\tau = (\mathbf{R}, \mathbf{F}, \mathbf{C}, \text{ari})$

Termenii lui \mathcal{L} , notați $\text{Trm}_{\mathcal{L}}$, sunt definiți inductiv astfel:

- orice variabilă este un termen;
- orice simbol de constantă este un termen;
- dacă $f \in \mathbf{F}$, $\text{ar}(f) = n$ și t_1, \dots, t_n sunt termeni, atunci $f(t_1, \dots, t_n)$ este termen.

Formulele atomice ale lui \mathcal{L} sunt definite astfel:

- dacă $R \in \mathbf{R}$, $\text{ar}(R) = n$ și t_1, \dots, t_n sunt termeni, atunci $R(t_1, \dots, t_n)$ este formulă atomică.

Formulele lui \mathcal{L} sunt definite astfel:

- orice formulă atomică este o formulă
- dacă φ este o formulă, atunci $\neg\varphi$ este o formulă
- dacă φ și ψ sunt formule, atunci $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \rightarrow \psi$ sunt formule
- dacă φ este o formulă și x este o variabilă, atunci $\forall x \varphi$, $\exists x \varphi$ sunt formule

Literali

- În calculul propozițional un **literal** este o **variabilă** sau **negația unei variabile**.

literal := $p \mid \neg p$ unde p este variabilă propozițională

Literali

- În calculul propozițional un **literal** este o **variabilă** sau **negația unei variabile**.

literal := $p \mid \neg p$ unde p este variabilă propozițională

- În logica de ordinul I un **literal** este o **formulă atomică** sau **negația unei formule atomice**.

literal := $P(t_1, \dots, t_n) \mid \neg P(t_1, \dots, t_n)$

unde $P \in \mathbf{R}$, $\text{ari}(P) = n$, și t_1, \dots, t_n sunt termeni.

Clauze

- O clauză este o disjuncție de literali.

Clauze

- O clauză este o disjuncție de literali.
- Dacă L_1, \dots, L_n sunt literali atunci clauza $L_1 \vee \dots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \dots, L_n\}$

clauză = mulțime de literali

Clauze

- O clauză este o disjuncție de literali.
- Dacă L_1, \dots, L_n sunt literali atunci clauza $L_1 \vee \dots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \dots, L_n\}$
$$\text{clauză} = \text{mulțime de literali}$$
- Clauza $C = \{L_1, \dots, L_n\}$ este satisfiabilă dacă $L_1 \vee \dots \vee L_n$ este satisfiabilă.

Clauze

- O clauză este o disjuncție de literali.
- Dacă L_1, \dots, L_n sunt literali atunci clauza $L_1 \vee \dots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \dots, L_n\}$
$$\text{clauză} = \text{mulțime de literali}$$
- Clauza $C = \{L_1, \dots, L_n\}$ este satisfiabilă dacă $L_1 \vee \dots \vee L_n$ este satisfiabilă.
- O clauză C este trivială dacă conține un literal și complementul lui.

Clauze

- O clauză este o disjuncție de literali.
- Dacă L_1, \dots, L_n sunt literali atunci clauza $L_1 \vee \dots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \dots, L_n\}$
$$\text{clauză} = \text{mulțime de literali}$$
- Clauza $C = \{L_1, \dots, L_n\}$ este satisfiabilă dacă $L_1 \vee \dots \vee L_n$ este satisfiabilă.
- O clauză C este trivială dacă conține un literal și complementul lui.
- Când $n = 0$ obținem clauza vidă, care se notează \square

Clauze

- O clauză este o disjuncție de literali.
- Dacă L_1, \dots, L_n sunt literali atunci clauza $L_1 \vee \dots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \dots, L_n\}$
$$\text{clauză} = \text{mulțime de literali}$$
- Clauza $C = \{L_1, \dots, L_n\}$ este satisfiabilă dacă $L_1 \vee \dots \vee L_n$ este satisfiabilă.
- O clauză C este trivială dacă conține un literal și complementul lui.
- Când $n = 0$ obținem clauza vidă, care se notează \square
- Prin definiție, clauza \square nu este satisfiabilă.

Clauze

- O clauză este o disjuncție de literali.
- Dacă L_1, \dots, L_n sunt literali atunci clauza $L_1 \vee \dots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \dots, L_n\}$
$$\text{clauză} = \text{mulțime de literali}$$
- Clauza $C = \{L_1, \dots, L_n\}$ este satisfiabilă dacă $L_1 \vee \dots \vee L_n$ este satisfiabilă.
- O clauză C este trivială dacă conține un literal și complementul lui.
- Când $n = 0$ obținem clauza vidă, care se notează \square
- Prin definiție, clauza \square nu este satisfiabilă.

Rezoluția este o metodă de verificare a satisfiabilității unei mulțimi de clauze.

Clauze în logica de ordinul I

$$\{\neg Q_1, \dots, \neg Q_n, P_1, \dots, P_k\}$$

unde $n, k \geq 0$ și $Q_1, \dots, Q_n, P_1, \dots, P_k$ sunt formule atomice.

- formula corespunzătoare este

$$\forall x_1 \dots \forall x_m (\neg Q_1 \vee \dots \vee \neg Q_n \vee P_1 \vee \dots \vee P_k)$$

unde x_1, \dots, x_m sunt toate variabilele care apar în clauză

- echivalent, putem scrie

$$\forall x_1 \dots \forall x_m (Q_1 \wedge \dots \wedge Q_n \rightarrow P_1 \vee \dots \vee P_k)$$

- cuantificarea universală a clauzelor este implicită

$$Q_1 \wedge \dots \wedge Q_n \rightarrow P_1 \vee \dots \vee P_k$$

Clauze definite. Programe logice. Clauze Horn

□ clauză:

$$\{\neg Q_1, \dots, \neg Q_n, P_1, \dots, P_k\} \text{ sau } Q_1 \wedge \dots \wedge Q_n \rightarrow P_1 \vee \dots \vee P_k$$

unde $n, k \geq 0$ și $Q_1, \dots, Q_n, P_1, \dots, P_k$ sunt formule atomice.

Clauze definite. Programe logice. Clauze Horn

- clauză:

$$\{\neg Q_1, \dots, \neg Q_n, P_1, \dots, P_k\} \text{ sau } Q_1 \wedge \dots \wedge Q_n \rightarrow P_1 \vee \dots \vee P_k$$

unde $n, k \geq 0$ și $Q_1, \dots, Q_n, P_1, \dots, P_k$ sunt formule atomice.

- clauză program definită: $k = 1$

- cazul $n > 0$: $Q_1 \wedge \dots \wedge Q_n \rightarrow P$

- cazul $n = 0$: $\top \rightarrow P$ (clauză unitate, fapt)

Program logic definit = mulțime finită de clauze definite

Clauze definite. Programe logice. Clauze Horn

- clauză:

$$\{\neg Q_1, \dots, \neg Q_n, P_1, \dots, P_k\} \text{ sau } Q_1 \wedge \dots \wedge Q_n \rightarrow P_1 \vee \dots \vee P_k$$

unde $n, k \geq 0$ și $Q_1, \dots, Q_n, P_1, \dots, P_k$ sunt formule atomice.

- clauză program definită: $k = 1$

 - cazul $n > 0$: $Q_1 \wedge \dots \wedge Q_n \rightarrow P$

 - cazul $n = 0$: $\top \rightarrow P$ (clauză unitate, fapt)

Program logic definit = mulțime finită de clauze definite

- clauză scop definită (țintă, întrebare): $k=0$

 - $Q_1 \wedge \dots \wedge Q_n \rightarrow \perp$

- clauza vidă □: $n = k = 0$

Clauze definite. Programe logice. Clauze Horn

- clauză:

$$\{\neg Q_1, \dots, \neg Q_n, P_1, \dots, P_k\} \text{ sau } Q_1 \wedge \dots \wedge Q_n \rightarrow P_1 \vee \dots \vee P_k$$

unde $n, k \geq 0$ și $Q_1, \dots, Q_n, P_1, \dots, P_k$ sunt formule atomice.

- clauză program definită: $k = 1$

- cazul $n > 0$: $Q_1 \wedge \dots \wedge Q_n \rightarrow P$

- cazul $n = 0$: $\top \rightarrow P$ (clauză unitate, fapt)

Program logic definit = mulțime finită de clauze definite

- clauză scop definită (țintă, întrebare): $k=0$

- $Q_1 \wedge \dots \wedge Q_n \rightarrow \perp$

- clauza vidă □: $n = k = 0$

Clauza Horn = clauză program definită sau clauză scop ($k \leq 1$)

Clauze Horn țintă

□ clauză scop definită (țintă, întrebare): $Q_1 \wedge \dots \wedge Q_n \rightarrow \perp$

- fie x_1, \dots, x_m toate variabilele care apar în Q_1, \dots, Q_n
 $\forall x_1 \dots \forall x_m (\neg Q_1 \vee \dots \vee \neg Q_n) \models \neg \exists x_1 \dots \exists x_m (Q_1 \wedge \dots \wedge Q_n)$
- clauza țintă o vom scrie Q_1, \dots, Q_n

Negația unei "întrebări" în PROLOG este clauză Horn țintă.

Programare logica

- Logica clauzelor definite/Logica Horn: un fragment al logicii de ordinul I în care singurele formule admise sunt clauze Horn
 - formule atomice: $P(t_1, \dots, t_n)$
 - $Q_1 \wedge \dots \wedge Q_n \rightarrow P$
unde toate Q_i, P sunt formule atomice, \top sau \perp

Programare logica

- Logica clauzelor definite/Logica Horn: un fragment al logicii de ordinul I în care singurele formule admise sunt clauze Horn
 - formule atomice: $P(t_1, \dots, t_n)$
 - $Q_1 \wedge \dots \wedge Q_n \rightarrow P$
unde toate Q_i, P sunt formule atomice, \top sau \perp
- Problema programării logice: reprezentăm cunoștințele ca o mulțime de clauze definite KB și suntem interesați să aflăm răspunsul la o întrebare de forma $Q_1 \wedge \dots \wedge Q_n$, unde toate Q_i sunt formule atomice

$$KB \models Q_1 \wedge \dots \wedge Q_n$$

- Variabilele din KB sunt **cuantificate universal**.
- Variabilele din Q_1, \dots, Q_n sunt **cuantificate existențial**.

Limbajul PROLOG are la bază logica clauzelor Horn.

Logica clauzelor definite

Exemplu

Fie următoarele clauze definite:

$\text{father}(\text{jon}, \text{ken}).$

$\text{father}(\text{ken}, \text{liz}).$

$\text{father}(X, Y) \rightarrow \text{ancestor}(X, Y)$

$\text{daughter}(X, Y) \rightarrow \text{ancestor}(Y, X)$

$\text{ancestor}(X, Y) \wedge \text{ancestor}(Y, Z) \rightarrow \text{ancestor}(X, Z)$

Putem întreba:

- $\text{ancestor}(\text{jon}, \text{liz})$
- dacă există Z astfel încât $\text{ancestor}(Z, \text{ken})$
(adică $\exists Z \text{ancestor}(Z, \text{ken})$)

Sistem de deducție pentru logica Horn

Sistem de deducție *backchain*

Sistem de deducție pentru clauze Horn

Pentru un program logic definit KB avem

Sistem de deducție *backchain*

Sistem de deducție pentru clauze Horn

Pentru un program logic definit KB avem

- Axiome: orice clauză din KB

Sistem de deducție *backchain*

Sistem de deducție pentru clauze Horn

Pentru un program logic definit KB avem

- **Axiome:** orice clauză din KB
- **Regula de deducție:** regula *backchain*

$$\frac{\theta(Q_1) \quad \theta(Q_2) \quad \dots \quad \theta(Q_n) \quad (Q_1 \wedge Q_2 \wedge \dots \wedge Q_n \rightarrow P)}{\theta(Q)}$$

unde $Q_1 \wedge Q_2 \wedge \dots \wedge Q_n \rightarrow P \in KB$, iar θ este cgu pentru Q și P .

Sistem de deducție

Exemplu

KB conține următoarele clauze definite:

father(jon, ken).

father(ken, liz).

father(X, Y) → ancestor(X, Y)

daughter(X, Y) → ancestor(Y, X)

ancestor(X, Y) ∧ ancestor(Y, Z) → ancestor(X, Z)

$$\frac{\theta(Q_1) \quad \theta(Q_2) \quad \dots \quad \theta(Q_n) \quad (Q_1 \wedge Q_2 \wedge \dots \wedge Q_n \rightarrow P)}{\theta(Q)}$$

unde $Q_1 \wedge Q_2 \wedge \dots \wedge Q_n \rightarrow P \in KB$, iar θ este cgu pentru Q și P

Sistem de deducție

Pentru o țintă Q , trebuie să găsim o clauză din KB

$$Q_1 \wedge \dots \wedge Q_n \rightarrow P,$$

și un unificator θ pentru Q și P . În continuare vom verifica $\theta(Q_1), \dots, \theta(Q_n)$.

Sistem de deducție

Pentru o țintă Q , trebuie să găsim o clauză din KB

$$Q_1 \wedge \dots \wedge Q_n \rightarrow P,$$

și un unificator θ pentru Q și P . În continuare vom verifica $\theta(Q_1), \dots, \theta(Q_n)$.

Exemplu

Pentru ținta

$$\text{ancestor}(\text{ken}, Z),$$

Sistem de deducție

Pentru o ţintă Q , trebuie să găsim o clauză din KB

$$Q_1 \wedge \dots \wedge Q_n \rightarrow P,$$

și un unificator θ pentru Q și P . În continuare vom verifica $\theta(Q_1), \dots, \theta(Q_n)$.

Exemplu

Pentru ţinta

$$\text{ancestor}(\text{ken}, Z),$$

putem folosi o clauză

$$\text{father}(Y, X) \rightarrow \text{ancestor}(Y, X)$$

cu unificatorul

$$\{Y/\text{ken}, X/Z\}$$

pentru a obține o nouă ţintă

$$\text{father}(\text{ken}, Z).$$

Sistem de deducție

$$\frac{\theta(Q_1) \quad \theta(Q_2) \quad \dots \quad \theta(Q_n) \quad (Q_1 \wedge Q_2 \wedge \dots \wedge Q_n \rightarrow P)}{\theta(Q)}$$

unde $Q_1 \wedge Q_2 \wedge \dots \wedge Q_n \rightarrow P \in KB$, iar θ este cgu pentru Q și P .

Exemplu

$$\frac{\begin{array}{c} \textit{father(ken, liz)} \\ \textit{father(ken, Z)} \end{array} \quad (\textit{father}(Y, X) \rightarrow \textit{ancestor}(Y, X))}{\textit{ancestor}(ken, Z)}$$

Puncte de decizie în programarea logica

Având doar această regulă, care sunt punctele de decizie în căutare?

Puncte de decizie în programarea logica

Având doar această regulă, care sunt punctele de decizie în căutare?

Ce clauză să alegem.

- Pot fi mai multe clauze a căror parte dreaptă se potrivește cu o întă.
- Aceasta este o alegere de tip **SAU**: este suficient ca oricare din variante să reușească.

Puncte de decizie în programarea logica

Având doar această regulă, care sunt punctele de decizie în căutare?

Ce clauză să alegem.

- Pot fi mai multe clauze a căror parte dreaptă se potrivește cu o țintă.
- Aceasta este o alegere de tip **SAU**: este suficient ca oricare din variante să reușească.

Ordinea în care rezolvăm noile ținte.

- Aceasta este o alegere de tip **ȘI**: trebuie arătate toate țintele noi.
- Ordinea în care le rezolvăm poate afecta găsirea unei derivări, depinzând de strategia de căutare folosită.

Strategia de căutare din Prolog

- Regula *backchain* conduce la un **sistem de deducție complet**:
Pentru o mulțime de clauze KB și o întă Q ,
dacă $KB \models Q$,
atunci există o derivare a lui Q folosind regula *backchain*.

Strategia de căutare din Prolog

- Regula *backchain* conduce la un **sistem de deducție complet**:
Pentru o mulțime de clauze KB și o întă Q ,
dacă $KB \models Q$,
atunci există o derivare a lui Q folosind regula *backchain*.
- Strategia de căutare din Prolog este de tip ***depth-first***,
 - **de sus în jos**
 - pentru alegerile de tip **SAU**
 - alege clauzele în ordinea în care apar în program
 - **de la stânga la dreapta**
 - pentru alegerile de tip **\$I**
 - alege noile înte în ordinea în care apar în clauza aleasă

Sistemul de inferență backchain

Notăm cu $KB \vdash_b Q$ dacă există o derivare a lui Q din KB folosind sistemul de inferență *backchain*.

Teoremă

Sistemul de inferență backchain este corect și complet pentru formule atomice fără variabile Q .

$$KB \models Q \quad \text{dacă și numai dacă} \quad KB \vdash_b Q$$

Sistemul de inferență backchain

Notăm cu $KB \vdash_b Q$ dacă există o derivare a lui Q din KB folosind sistemul de inferență *backchain*.

Teoremă

Sistemul de inferență backchain este corect și complet pentru formule atomice fără variabile Q .

$$KB \models Q \quad \text{dacă și numai dacă} \quad KB \vdash_b Q$$

Sistemul de inferență backchain este corect și complet și pentru formule atomice cu variabile Q :

$$KB \models \exists x Q(x) \quad \text{dacă și numai dacă} \quad KB \vdash_b \theta(Q)$$

pentru o substituție θ .

Rezoluție SLD

Regula *backchain* și rezoluția SLD

- Regula *backchain* este implementată în programarea logică prin rezoluția SLD (Selected, Linear, Definite).
- Prolog are la bază rezoluția SLD.

Rezoluția SLD

Fie KB o mulțime de clauze definite.

SLD

$$\frac{\neg Q_1 \vee \cdots \vee \neg Q_i \vee \cdots \vee \neg Q_n}{\theta(\neg Q_1 \vee \cdots \vee \neg P_1 \vee \cdots \vee \neg P_m \vee \cdots \vee \neg Q_n)}$$

unde

- $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ este o clauză definită din KB (în care toate variabilele au fost redenumite) și
- variabilele din $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ și Q_i se redenumesc
- θ este c.g.u pentru Q_i și Q

Rezoluția SLD

Exemplu

father(eddard,sansa).

father(eddard,jonSnow).

stark(eddard).

?- stark(jonSnow)

stark(catelyn).

stark(X) :- father(Y,X), stark(Y).

SLD

$$\frac{\neg Q_1 \vee \dots \vee \neg Q_i \vee \dots \vee \neg Q_n}{\theta(\neg Q_1 \vee \dots \vee \neg P_1 \vee \dots \vee \neg P_m \vee \dots \vee \neg Q_n)}$$

- $Q \vee \neg P_1 \vee \dots \vee \neg P_m$ este o clauză definită din KB
- variabilele din $Q \vee \neg P_1 \vee \dots \vee \neg P_m$ și Q_i se redenumesc
- θ este c.g.u pentru Q_i și Q .

Rezoluția SLD

Exemplu

father(eddard, sansa)

father(eddard, jonSnow)

$\neg \text{stark(jonSnow)}$

stark(eddard)

stark(catelyn)

$\theta(X) = \text{jonSnow}$

stark(X) $\vee \neg \text{father}(Y, X) \vee \neg \text{stark}(Y)$

SLD

$$\frac{\neg Q_1 \vee \dots \vee \neg Q_i \vee \dots \vee \neg Q_n}{\theta(\neg Q_1 \vee \dots \vee \neg P_1 \vee \dots \vee \neg P_m \vee \dots \vee \neg Q_n)}$$

- $Q \vee \neg P_1 \vee \dots \vee \neg P_m$ este o clauză definită din KB
- variabilele din $Q \vee \neg P_1 \vee \dots \vee \neg P_m$ și Q_i se redenumesc
- θ este c.g.u pentru Q_i și Q .

Rezoluția SLD

Exemplu

$father(eddard, sansa)$

$father(eddard, jonSnow)$

$stark(eddard)$

$stark(catelyn)$

$\neg stark(jonSnow)$

$\neg father(Y, jonSnow) \vee \neg stark(Y)$

$\theta(X) = jonSnow$

$stark(X) \vee \neg father(Y, X) \vee \neg stark(Y)$

SLD

$$\frac{\neg Q_1 \vee \dots \vee \neg Q_i \vee \dots \vee \neg Q_n}{\theta(\neg Q_1 \vee \dots \vee \neg P_1 \vee \dots \vee \neg P_m \vee \dots \vee \neg Q_n)}$$

- $Q \vee \neg P_1 \vee \dots \vee \neg P_m$ este o clauză definită din KB
- variabilele din $Q \vee \neg P_1 \vee \dots \vee \neg P_m$ și Q_i se redenumesc
- θ este c.g.u pentru Q_i și Q .

Rezoluția SLD

Exemplu

$\text{father}(eddard, sansa)$

$\text{father}(eddard, jonSnow)$

$\text{stark}(eddard)$

$\text{stark}(catelyn)$

$\text{stark}(X) \vee \neg \text{father}(Y, X) \vee \neg \text{stark}(Y)$

$$\frac{\neg \text{stark}(jonSnow)}{\neg \text{father}(Y, jonSnow) \vee \neg \text{stark}(Y)}$$

Rezoluția SLD

Exemplu

$\text{father}(eddard, sansa)$

$\text{father}(eddard, \text{jonSnow})$

$\text{stark}(eddard)$

$\text{stark}(\text{catelyn})$

$\text{stark}(X) \vee \neg \text{father}(Y, X) \vee \neg \text{stark}(Y)$

$$\frac{\neg \text{stark}(\text{jonSnow})}{\neg \text{father}(Y, \text{jonSnow}) \vee \neg \text{stark}(Y)}$$

$$\frac{\neg \text{father}(Y, \text{jonSnow}) \vee \neg \text{stark}(Y)}{\neg \text{stark}(eddard)}$$

Rezoluția SLD

Exemplu

$\text{father}(eddard, sansa)$

$\text{father}(eddard, \text{jonSnow})$

$\text{stark}(eddard)$

$\text{stark}(\text{catelyn})$

$\text{stark}(X) \vee \neg \text{father}(Y, X) \vee \neg \text{stark}(Y)$

$$\frac{\neg \text{stark}(\text{jonSnow})}{\neg \text{father}(Y, \text{jonSnow}) \vee \neg \text{stark}(Y)}$$

$$\frac{\neg \text{father}(Y, \text{jonSnow}) \vee \neg \text{stark}(Y)}{\neg \text{stark}(eddard)}$$

$$\frac{\neg \text{stark}(eddard)}{\square}$$

Rezoluția SLD

Fie KB o mulțime de clauze definite și $Q_1 \wedge \dots \wedge Q_m$ o întrebare, unde Q_i sunt formule atomice.

- O derivare din KB prin rezoluție SLD este o secvență

$$G_0 := \neg Q_1 \vee \dots \vee \neg Q_m, \quad G_1, \quad \dots, \quad G_k, \quad \dots$$

în care G_{i+1} se obține din G_i prin regula **SLD**.

- Dacă există un k cu $G_k = \square$ (clauza vidă), atunci derivarea se numește **SLD-respingere**.

Rezoluția SLD

Teoremă (Completitudinea SLD-rezoluției)

Sunt echivalente:

- există o *SLD-respingere* a lui $Q_1 \wedge \dots \wedge Q_m$ din KB ,
- $KB \vdash_b Q_1 \wedge \dots \wedge Q_m$,
- $KB \models Q_1 \wedge \dots \wedge Q_m$.

Rezoluția SLD

Teoremă (Completitudinea SLD-rezoluției)

Sunt echivalente:

- există o **SLD-respingere** a lui $Q_1 \wedge \dots \wedge Q_m$ din KB ,
- $KB \vdash_b Q_1 \wedge \dots \wedge Q_m$,
- $KB \models Q_1 \wedge \dots \wedge Q_m$.

Demonstrație

Rezultă din completitudinea sistemului de deducție backchain și din faptul că:

există o **SLD-respingere** a lui $Q_1 \wedge \dots \wedge Q_m$ din KB

ddacă

$KB \vdash_b Q_1 \wedge \dots \wedge Q_m$

□

Rezoluția SLD - arbori de căutare

Arbore SLD

- Presupunem că avem o mulțime de clauze definite KB și o țintă
$$G_0 = \neg Q_1 \vee \dots \vee \neg Q_m$$
- Construim un arbore de căutare ([arbore SLD](#)) astfel:
 - Fiecare nod al arborelui este o țintă (posibil vidă)
 - Rădăcina este G_0
 - Dacă arborele are un nod G_i , iar G_{i+1} se obține din G_i folosind regula SLD folosind o clauză $C_i \in KB$, atunci nodul G_i are copilul G_{i+1} . Muchia dintre G_i și G_{i+1} este etichetată cu C_i .
- Dacă un arbore SLD cu rădăcina G_0 are o frunză \square (clauza vidă), atunci există o SLD-respingere a lui G_0 din KB .

Rezoluția SLD

Exemplu

□ Fie KB următoarea mulțime de clauze definite:

- 1 $\text{grandfather}(X, Z) : \neg \text{father}(X, Y), \text{parent}(Y, Z)$
- 2 $\text{parent}(X, Y) : \neg \text{father}(X, Y)$
- 3 $\text{parent}(X, Y) : \neg \text{mother}(X, Y)$
- 4 $\text{father}(\text{ken}, \text{diana})$
- 5 $\text{mother}(\text{diana}, \text{brian})$

□ Găsiți o respingere din KB pentru

$: \neg \text{grandfather}(\text{ken}, Y)$

Rezoluția SLD

Exemplu

□ Fie KB următoarea mulțime de clauze definite:

- 1 $\text{grandfather}(X, Z) \vee \neg\text{father}(X, Y) \vee \neg\text{parent}(Y, Z)$
- 2 $\text{parent}(X, Y) \vee \neg\text{father}(X, Y)$
- 3 $\text{parent}(X, Y) \vee \neg\text{mother}(X, Y)$
- 4 $\text{father}(\text{ken}, \text{diana})$
- 5 $\text{mother}(\text{diana}, \text{brian})$

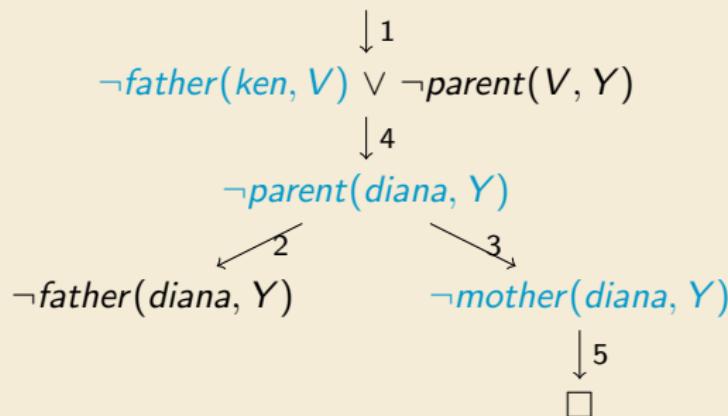
□ Găsiți o respingere din KB pentru

$\neg\text{grandfather}(\text{ken}, Y)$

Rezoluția SLD

Exemplu

- 1 $\text{grandfather}(X, Z) \vee \neg\text{father}(X, Y) \vee \neg\text{parent}(Y, Z)$
- 2 $\text{parent}(X, Y) \vee \neg\text{father}(X, Y)$
- 3 $\text{parent}(X, Y) \vee \neg\text{mother}(X, Y)$
- 4 $\text{father}(\text{ken}, \text{diana})$
- 5 $\text{mother}(\text{diana}, \text{brian}) \quad \neg\text{grandfather}(\text{ken}, Y)$



Rezoluția SLD

Exemplu

2 $\text{parent}(X, Y) \vee \neg\text{father}(X, Y)$

$\neg\text{parent}(\text{diana}, Y)$

$\neg\text{father}(\text{diana}, Y)$



Aplicarea SLD:

Rezoluția SLD

Exemplu

2 $\text{parent}(X, Y) \vee \neg\text{father}(X, Y)$

$$\frac{\neg\text{parent}(\text{diana}, Y)}{\neg\text{father}(\text{diana}, Y)}$$

↙
2

Aplicarea SLD:

- redenumesc variabilele: $\text{parent}(X, Y_2) \vee \neg\text{father}(X, Y_2)$
- determin unificatorul: $\theta = X/\text{diana}, Y_2/Y$
- aplic regula:
$$\frac{\neg\text{parent}(\text{diana}, Y)}{\neg\text{father}(\text{diana}, Y)}$$

Limbajul Prolog

- Am arătat că **sistemul de inferență din spatele Prolog-ului este complet.**
 - Dacă o întrebare este consecință logică a unei mulțimi de clauze, atunci există o derivare a întrebării.
- Totuși, **strategia de căutare din Prolog este incompletă!**
 - Chiar dacă o întrebare este consecință logică a unei mulțimi de clauze, Prolog nu găsește mereu o derivare a întrebării.

Limbajul Prolog

Exemplu

```
warmerClimate :- albedoDecrease.  
warmerClimate :- carbonIncrease.  
iceMelts :- warmerClimate.  
albedoDecrease :- iceMelts.  
carbonIncrease.  
  
?- iceMelts.  
! Out of local stack
```

Limbajul Prolog

Exemplu

```
warmerClimate :- albedoDecrease.  
warmerClimate :- carbonIncrease.  
iceMelts :- warmerClimate.  
albedoDecrease :- iceMelts.  
carbonIncrease.  
  
?- iceMelts.  
! Out of local stack
```

Limbajul Prolog

Exemplu (cont.)

Există o derivare a lui *iceMelts* în sistemul de deducție din clauzele:

<i>albedoDecrease</i>	\rightarrow	<i>warmerClimate</i>
<i>carbonIncrease</i>	\rightarrow	<i>warmerClimate</i>
<i>warmerClimate</i>	\rightarrow	<i>iceMelts</i>
<i>iceMelts</i>	\rightarrow	<i>albedoDecrease</i>
T	\rightarrow	<i>carbonIncrease</i>

<i>carbonInc.</i>	$\frac{\text{carbonInc. } \quad \text{carbonInc.} \rightarrow \text{warmerClim.}}{\text{warmerClim.}}$	$\text{warmerClim.} \rightarrow \text{iceMelts}$
		<hr/>
		<i>iceMelts</i>

Rezoluția SLD - arbori de căutare

Exercițiu

Desenați arborele SLD pentru programul Prolog de mai jos și ținta
?- p(X,X).

- | | |
|------------------------------|--------------------|
| 1. p(X,Y) :- q(X,Z), r(Z,Y). | 7. s(X) :- t(X,a). |
| 2. p(X,X) :- s(X). | 8. s(X) :- t(X,b). |
| 3. q(X,b). | 9. s(X) :- t(X,X). |
| 4. q(b,a). | 10. t(a,b). |
| 5. q(X,a) :- r(a,X). | 11. t(b,a). |
| 6. r(b,a). | |

Rezoluția SLD - arbori de căutare

1. $p(X, Y) :- q(X, Z), r(Z, Y).$
 2. $p(X, X) :- s(X).$
 3. $q(X, b).$

$p(X, Y) \vee \neg q(X, Z) \vee \neg r(Z, Y)$
 $p(X, X) \vee \neg s(X)$
 $q(X, b)$

4. $q(b, a).$
 5. $q(X, a) :- r(a, X).$
 6. $r(b, a).$

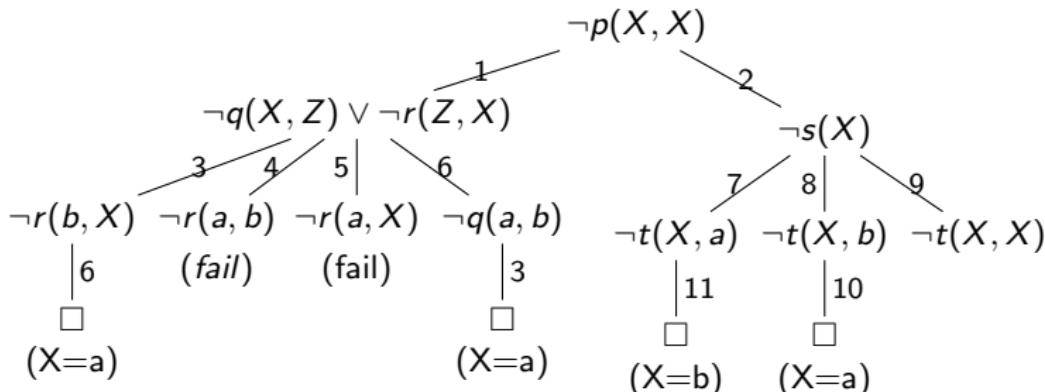
$q(b, a)$
 $q(X, a) \vee \neg r(a, X)$
 $r(b, a)$

7. $s(X) :- t(X, a).$
 8. $s(X) :- t(X, b).$
 9. $s(X) :- t(X, X).$

$s(X) \vee \neg t(X, a)$
 $s(X) \vee \neg t(X, b)$
 $s(X) \vee \neg t(X, X)$

10. $t(a, b).$
 11. $t(b, a).$

$t(a, b)$
 $t(b, a)$





Pe săptămâna viitoare!

Curs 9

2021-2022

Fundamentele limbajelor de programare

Cuprins

- 1 Limbajul IMP**
- 2 Semantica programelor - idei generale**
- 3 Semantica operațională small-step**

Limbajul IMP

Limbajul IMP

Vom implementa un limbaj care conține:

- Expresii**

- Aritmetice
 - Booleene

$$\begin{array}{l} x + 3 \\ x \geq 7 \end{array}$$

Limbajul IMP

Vom implementa un limbaj care conține:

- Expresii**

- Aritmetice
- Booleene

$x + 3$
 $x \geq 7$

- Instructiuni**

- De atribuire

$x = 5$

Limbajul IMP

Vom implementa un limbaj care conține:

- Expresii**

- Aritmetice
- Booleene

$x + 3$
 $x \geq 7$

- Instructiuni**

- De atribuire
- Conditionale

$x = 5$
`if(x >= 7, x = 5, x = 0)`

Limbajul IMP

Vom implementa un limbaj care conține:

- Expresii**

- Aritmetice
- Booleene

$x + 3$
 $x \geq 7$

- Instructiuni**

- De atribuire
- Conditionale
- De ciclare

$x = 5$
`if(x >= 7, x = 5, x = 0)`
`while(x >= 7, x = x - 1)`

Limbajul IMP

Vom implementa un limbaj care conține:

- **Expresii**

- Aritmetice
 - Booleene

$x + 3$
 $x \geq 7$

- **Instructiuni**

- De atribuire
 - Condiționale
 - De ciclare

$x = 5$
if($x \geq 7$, $x = 5$, $x = 0$)
while($x \geq 7$, $x = x - 1$)

- **Compunerea instrucțiunilor**

$x=7$; while($x \geq 0$, $x=x-1$)

Limbajul IMP

Vom implementa un limbaj care conține:

- **Expresii**

- Aritmetice
 - Booleene

$x + 3$
 $x \geq 7$

- **Instructiuni**

- De atribuire
 - Condiționale
 - De ciclare

$x = 5$
if($x \geq 7$, $x = 5$, $x = 0$)
while($x \geq 7$, $x = x - 1$)

- **Componerea instrucțiunilor**

$x=7; \text{ while}(x>=0, x=x-1)$

- **Blocuri de instrucțiuni**

{ $x=7; \text{ while}(x>=0, x=x-1)$ }

Limbajul IMP

Exemplu

Un program în limbajul IMP

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

□ Semantica

după executia programului, se evaluează sum

Sintaxa BNF a limbajului IMP

$E ::= n \mid x$
| $E + E$ | $E - E$ | $E * E$

$B ::= \text{true} \mid \text{false}$
| $E <= E$ | $E >= E$ | $E == E$
| $\text{not}(B)$ | $\text{and}(B, B)$ | $\text{or}(B, B)$

$C ::= \text{skip}$
| $x = E$
| $\text{if}(B, C, C)$
| $\text{while}(B, C)$
| $\{C\}$ | $C ; C$

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

Semantica programelor - idei generale

Ce înseamnă semantica formală?

Ce definește un limbaj de programare?

Ce înseamnă semantica formală?

Ce definește un limbaj de programare?

- **Sintaxa** – Simboluri de operație, cuvinte cheie, descriere (formală) a programelor/expresiilor bine formate

Ce înseamnă semantica formală?

Ce definește un limbaj de programare?

- **Sintaxa** – Simboluri de operație, cuvinte cheie, descriere (formală) a programelor/expresiilor bine formate
- **Practic** – Un limbaj e definit de modul cum poate fi folosit
 - Manual de utilizare și exemple de bune practici
 - Implementare (compilator/interpreter)
 - Instrumente ajutătoare (analizor de sintaxă, depanator)

Ce înseamnă semantica formală?

Ce definește un limbaj de programare?

- **Sintaxa** – Simboluri de operație, cuvinte cheie, descriere (formală) a programelor/expresiilor bine formate
- **Practic** – Un limbaj e definit de modul cum poate fi folosit
 - Manual de utilizare și exemple de bune practici
 - Implementare (compilator/interpreter)
 - Instrumente ajutătoare (analizor de sintaxă, depanator)
- **Semantica** – Ce înseamnă/care e comportamentul unei instrucțiuni?

La ce folosește semantica?

- Să înțelegem un limbaj în profunzime
 - Ca programator: pe ce mă pot baza când programez în limbajul dat
 - Ca implementator al limbajului: ce garanții trebuie să ofer

La ce folosește semantica?

- Să înțelegem un limbaj în profunzime
 - Ca programator: pe ce mă pot baza când programez în limbajul dat
 - Ca implementator al limbajului: ce garanții trebuie să ofer
- Ca instrument în proiectarea unui nou limbaj/a unei extensii
 - Înțelegerea componentelor și a relațiilor dintre ele
 - Exprimarea (și motivarea) deciziilor de proiectare
 - Demonstrarea unor proprietăți generice ale limbajului

La ce folosește semantica?

- Să înțelegem un limbaj în profunzime
 - Ca programator: pe ce mă pot baza când programez în limbajul dat
 - Ca implementator al limbajului: ce garanții trebuie să ofer
- Ca instrument în proiectarea unui nou limbaj/a unei extensii
 - Înțelegerea componentelor și a relațiilor dintre ele
 - Exprimarea (și motivarea) deciziilor de proiectare
 - Demonstrarea unor proprietăți generice ale limbajului
- Ca bază pentru demonstrarea corectitudinii programelor

Tipuri de semantică

- Limbaj natural – descriere textuală a efectelor

Tipuri de semantică

- **Limbaj natural** – descriere textuală a efectelor
- **Axiomatică** – descrierea folosind logică a efectelor unei instrucțiuni
 - $\vdash \{\varphi\} \text{cod}\{\psi\}$
 - modelează un program prin formulele logice pe care le satisface
 - utilă pentru demonstrarea corectitudinii

Tipuri de semantică

- **Limbaj natural** – descriere textuală a efectelor
- **Axiomatică** – descrierea folosind logică a efectelor unei instrucțiuni
 - $\vdash \{\varphi\} cod\{\psi\}$
 - modelează un program prin formulele logice pe care le satisface
 - utilă pentru demonstrarea corectitudinii
- **Denotațională** – asocierea unui obiect matematic (denotație)
 - $[\![cod]\!]$
 - modelează un program ca obiecte matematice
 - utilă pentru fundamente matematice

Tipuri de semantică

- **Limbaj natural** – descriere textuală a efectelor
- **Axiomatică** – descrierea folosind logică a efectelor unei instrucțiuni
 - $\vdash \{\varphi\} cod\{\psi\}$
 - modelează un program prin formulele logice pe care le satisface
 - utilă pentru demonstrarea corectitudinii
- **Denotațională** – asocierea unui obiect matematic (denotație)
 - $[\![cod]\!]$
 - modelează un program ca obiecte matematice
 - utilă pentru fundamentele matematice
- **Operatională** – asocierea unei demonstrații pentru execuție
 - $\langle cod, \sigma \rangle \rightarrow \langle cod', \sigma' \rangle$
 - modelează un program prin execuția pe o mașină abstractă
 - utilă pentru implementarea de compilatoare și interprotoare

Tipuri de semantică

- **Limbaj natural** – descriere textuală a efectelor
- **Axiomatică** – descrierea folosind logică a efectelor unei instrucțiuni
 - $\vdash \{\varphi\} cod\{\psi\}$
 - modelează un program prin formulele logice pe care le satisface
 - utilă pentru demonstrarea corectitudinii
- **Denotațională** – asocierea unui obiect matematic (denotație)
 - $[\![cod]\!]$
 - modelează un program ca obiecte matematice
 - utilă pentru fundamentele matematice
- **Operatională** – asocierea unei demonstrații pentru execuție
 - $\langle cod, \sigma \rangle \rightarrow \langle cod', \sigma' \rangle$
 - modelează un program prin execuția pe o mașină abstractă
 - utilă pentru implementarea de compilatoare și interprotoare
- **Statică** – asocierea unui sistem de tipuri care exclude programe eronate

Semantica operatională small-step

Imagine de ansamblu

- Semantica operațională descrie cum se execută un program pe o masină abstractă (ideală).

Imagine de ansamblu

- Semantica operațională descrie cum se execută un program pe o masină abstractă (ideală).
- **Semantica operațională small-step**
 - semantica structurală, a pa silor mici
 - descrie cum o execuție a programului avansează în funcție de reduceri succesive.

$$\langle cod, \sigma \rangle \rightarrow \langle cod', \sigma' \rangle$$

Imagine de ansamblu

- Semantica operațională descrie cum se execută un program pe o masină abstractă (ideală).
- **Semantica operațională small-step**
 - semantica structurală, a pa silor mici
 - descrie cum o execuție a programului avansează în funcție de reduceri succesive.

$$\langle cod, \sigma \rangle \rightarrow \langle cod', \sigma' \rangle$$

- **Semantica operațională big-step**
 - semantică naturală, într-un pas mare

Starea execuției

- Starea execuției unui program IMP la un moment dat este dată de valorile deținute în acel moment de variabilele declarate în program.
- Formal, starea execuției unui program IMP la un moment dat este o **funcție parțială** (cu domeniu finit):

$$\sigma : \text{Var} \rightharpoonup \text{Int}$$

Starea execuției

- Starea execuției unui program IMP la un moment dat este dată de valorile deținute în acel moment de variabilele declarate în program.
- Formal, starea execuției unui program IMP la un moment dat este o **funcție parțială** (cu domeniu finit):

$$\sigma : \text{Var} \rightharpoonup \text{Int}$$

□ Notatii:

- Descrierea funcției prin enumerare: $\sigma = n \mapsto 10, sum \mapsto 0$
- Funcția vidă \perp , nedefinită pentru nicio variabilă
- Obținerea valorii unei variabile: $\sigma(x)$
- Suprascrierea valorii unei variabile:

$$\sigma_{x \leftarrow v}(y) = \begin{cases} \sigma(y), & \text{dacă } y \neq x \\ v, & \text{dacă } y = x \end{cases}$$

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
 - Semantică Operațională Structurală
 - semantică prin tranziții
 - semantică prin reducere
- Definește cel mai mic pas de execuție ca o relație „de tranziție” între configurații:

$$\langle cod , \sigma \rangle \rightarrow \langle cod' , \sigma' \rangle$$

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
 - Semantică Operațională Structurală
 - semantică prin tranzitii
 - semantică prin reducere
- Definește cel mai mic pas de execuție ca o relație „de tranzitie” între configurații:
$$\langle cod , \sigma \rangle \rightarrow \langle cod' , \sigma' \rangle$$
- Execuția se obține ca o succesiune de astfel de tranzitii:

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
 - Semantică Operațională Structurală
 - semantică prin tranzitii
 - semantică prin reducere
- Definește cel mai mic pas de execuție ca o relație „de tranzitie” între configurații:

$$\langle \text{cod} , \sigma \rangle \rightarrow \langle \text{cod}' , \sigma' \rangle$$

- Execuția se obține ca o succesiune de astfel de tranzitii:

$$\langle x = 0 ; x = x + 1 ; , \perp \rangle$$

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
 - Semantică Operațională Structurală
 - semantică prin tranzitii
 - semantică prin reducere
- Definește cel mai mic pas de execuție ca o relație „de tranzitie” între configurații:

$$\langle \text{cod} , \sigma \rangle \rightarrow \langle \text{cod}' , \sigma' \rangle$$

- Execuția se obține ca o succesiune de astfel de tranzitii:

$$\langle x = 0 ; x = x + 1 ; , \perp \rangle \rightarrow \langle x = x + 1 ; , x \mapsto 0 \rangle$$

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
 - Semantică Operațională Structurală
 - semantică prin tranzitii
 - semantică prin reducere
- Definește cel mai mic pas de execuție ca o relație „de tranzitie” între configurații:

$$\langle \text{cod} , \sigma \rangle \rightarrow \langle \text{cod}' , \sigma' \rangle$$

- Execuția se obține ca o succesiune de astfel de tranzitii:

$$\begin{aligned}\langle x = 0 ; x = x + 1 ; , \perp \rangle &\rightarrow \langle x = x + 1 ; , x \mapsto 0 \rangle \\ &\rightarrow \langle x = 0 + 1 ; , x \mapsto 0 \rangle\end{aligned}$$

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
 - Semantică Operațională Structurală
 - semantică prin tranzitii
 - semantică prin reducere
- Definește cel mai mic pas de execuție ca o relație „de tranzitie” între configurații:

$$\langle \text{cod} , \sigma \rangle \rightarrow \langle \text{cod}' , \sigma' \rangle$$

- Execuția se obține ca o succesiune de astfel de tranzitii:

$$\begin{aligned}\langle x = 0 ; x = x + 1 ; , \perp \rangle &\rightarrow \langle x = x + 1 ; , x \mapsto 0 \rangle \\ &\rightarrow \langle x = 0 + 1 ; , x \mapsto 0 \rangle \\ &\rightarrow \langle x = 1 ; , x \mapsto 0 \rangle\end{aligned}$$

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
 - Semantică Operațională Structurală
 - semantică prin tranzitii
 - semantică prin reducere
- Definește cel mai mic pas de execuție ca o relație „de tranzitie” între configurații:

$$\langle \text{cod} , \sigma \rangle \rightarrow \langle \text{cod}' , \sigma' \rangle$$

- Execuția se obține ca o succesiune de astfel de tranzitii:

$$\begin{aligned}\langle x = 0 ; x = x + 1 ; , \perp \rangle &\rightarrow \langle x = x + 1 ; , x \mapsto 0 \rangle \\ &\rightarrow \langle x = 0 + 1 ; , x \mapsto 0 \rangle \\ &\rightarrow \langle x = 1 ; , x \mapsto 0 \rangle \\ &\rightarrow \langle \{ \} , x \mapsto 1 \rangle\end{aligned}$$

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
 - Semantică Operațională Structurală
 - semantică prin tranzitii
 - semantică prin reducere
- Definește cel mai mic pas de execuție ca o relație „de tranzitie” între configurații:

$$\langle \text{cod} , \sigma \rangle \rightarrow \langle \text{cod}' , \sigma' \rangle$$

- Execuția se obține ca o succesiune de astfel de tranzitii:

$$\begin{aligned}\langle x = 0 ; x = x + 1 ; , \perp \rangle &\rightarrow \langle x = x + 1 ; , x \mapsto 0 \rangle \\ &\rightarrow \langle x = 0 + 1 ; , x \mapsto 0 \rangle \\ &\rightarrow \langle x = 1 ; , x \mapsto 0 \rangle \\ &\rightarrow \langle \{ \} , x \mapsto 1 \rangle\end{aligned}$$

- Cum definim această relație?

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
 - Semantică Operațională Structurală
 - semantică prin tranzitii
 - semantică prin reducere
- Definește cel mai mic pas de execuție ca o relație „de tranzitie” între configurații:

$$\langle \text{cod} , \sigma \rangle \rightarrow \langle \text{cod}' , \sigma' \rangle$$

- Execuția se obține ca o succesiune de astfel de tranzitii:

$$\begin{aligned}\langle x = 0 ; x = x + 1 ; , \perp \rangle &\rightarrow \langle x = x + 1 ; , x \mapsto 0 \rangle \\ &\rightarrow \langle x = 0 + 1 ; , x \mapsto 0 \rangle \\ &\rightarrow \langle x = 1 ; , x \mapsto 0 \rangle \\ &\rightarrow \langle \{ \} , x \mapsto 1 \rangle\end{aligned}$$

- Cum definim această relație? Prin inducție după elementele din sintaxă.

Redex. Reguli structurale. Axiome

□ Expresie reductibilă (redex)

- Fragmentul de sintaxă care va fi procesat la pasul următor

if ($0 \leq 5 + 7 * x$, $r = 1$, $r = 0$)

Redex. Reguli structurale. Axiome

□ Expresie reductibilă (redex)

- Fragmentul de sintaxă care va fi procesat la pasul următor

if ($0 \leq 5 + 7 * \textcolor{red}{x}$, $r = 1$, $r = 0$)

Redex. Reguli structurale. Axiome

□ Expresie reductibilă (redex)

- Fragmentul de sintaxă care va fi procesat la pasul următor

`if (0 <= 5 + 7 * x , r = 1 , r = 0)`

□ Reguli structurale

- Folosesc la identificarea următorului redex
- Definite recursiv pe structura termenilor

Redex. Reguli structurale. Axiome

□ Expresie reductibilă (redex)

- ❑ Fragmentul de sintaxă care va fi procesat la pasul următor

`if (0 <= 5 + 7 * x , r = 1 , r = 0)`

□ Reguli structurale

- ❑ Folosesc la identificarea următorului redex
- ❑ Definite recursiv pe structura termenilor

$$\frac{\langle b , \sigma \rangle \rightarrow \langle b' , \sigma \rangle}{\langle \text{if} (b, bl_1, bl_2) , \sigma \rangle \rightarrow \langle \text{if} (b', bl_1, bl_2) , \sigma \rangle}$$

Redex. Reguli structurale. Axiome

□ Expresie reductibilă (redex)

- ❑ Fragmentul de sintaxă care va fi procesat la pasul următor

`if (0 <= 5 + 7 * x , r = 1 , r = 0)`

□ Reguli structurale

- ❑ Folosesc la identificarea următorului redex
- ❑ Definite recursiv pe structura termenilor

$$\frac{\langle b , \sigma \rangle \rightarrow \langle b' , \sigma \rangle}{\langle \text{if} (b, bl_1, bl_2) , \sigma \rangle \rightarrow \langle \text{if} (b', bl_1, bl_2) , \sigma \rangle}$$

□ Axiome

- ❑ Realizează pasul computațional

Redex. Reguli structurale. Axiome

□ Expresie reductibilă (redex)

- ❑ Fragmentul de sintaxă care va fi procesat la pasul următor

$\text{if } (0 \leq 5 + 7 * \textcolor{red}{x} , r = 1 , r = 0)$

□ Reguli structurale

- ❑ Folosesc la identificarea următorului redex
- ❑ Definite recursiv pe structura termenilor

$$\frac{\langle b , \sigma \rangle \rightarrow \langle b' , \sigma \rangle}{\langle \text{if } (b, bl_1, bl_2) , \sigma \rangle \rightarrow \langle \text{if } (b', bl_1, bl_2) , \sigma \rangle}$$

□ Axiome

- ❑ Realizează pasul computațional

$$\langle \text{if } (\text{true}, bl_1, bl_2) , \sigma \rangle \rightarrow \langle bl_1 , \sigma \rangle$$

Semantica expresiilor aritmetice

- Semantica unui întreg este o valoare
 - nu poate fi redex, deci nu avem regulă

Semantica expresiilor aritmetice

- Semantica unui întreg este o valoare
 - nu poate fi redex, deci nu avem regulă
- Semantica unei variabile
 - (Id) $\langle x, \sigma \rangle \rightarrow \langle i, \sigma \rangle$ dacă $\sigma(x) = i$

Semantica expresiilor aritmetice

- Semantica unui întreg este o valoare
 - nu poate fi redex, deci nu avem regulă

- Semantica unei variabile

(Id) $\langle x , \sigma \rangle \rightarrow \langle i , \sigma \rangle$ dacă $\sigma(x) = i$

- Semantica adunării a două expresii aritmetice

(Add) $\langle i_1 + i_2 , \sigma \rangle \rightarrow \langle i , \sigma \rangle$ dacă $i_1 + i_2 = i$

Semantica expresiilor aritmetice

- Semantica unui întreg este o valoare
 - nu poate fi redex, deci nu avem regulă

- Semantica unei variabile

(Id) $\langle x , \sigma \rangle \rightarrow \langle i , \sigma \rangle$ dacă $\sigma(x) = i$

- Semantica adunării a două expresii aritmetice

(Add) $\langle i_1 + i_2 , \sigma \rangle \rightarrow \langle i , \sigma \rangle$ dacă $i_1 + i_2 = i$

$$\frac{\langle a_1 , \sigma \rangle \rightarrow \langle a'_1 , \sigma \rangle}{\langle a_1 + a_2 , \sigma \rangle \rightarrow \langle a'_1 + a_2 , \sigma \rangle} \qquad \frac{\langle a_2 , \sigma \rangle \rightarrow \langle a'_2 , \sigma \rangle}{\langle a_1 + a_2 , \sigma \rangle \rightarrow \langle a_1 + a'_2 , \sigma \rangle}$$

Observatie: ordinea de evaluare a argumentelor este nespecificată.

Semantica expresiilor booleene

□ Semantica operatorului de comparație

(**LEQ-FALSE**) $\langle i_1 = < i_2 , \sigma \rangle \rightarrow \langle \text{false} , \sigma \rangle$ dacă $i_1 > i_2$

(**LEQ-TRUE**) $\langle i_1 = < i_2 , \sigma \rangle \rightarrow \langle \text{true} , \sigma \rangle$ dacă $i_1 \leq i_2$

Semantica expresiilor booleene

□ Semantica operatorului de comparație

(**LEQ-FALSE**) $\langle i_1 = < i_2 , \sigma \rangle \rightarrow \langle \text{false} , \sigma \rangle$ dacă $i_1 > i_2$

(**LEQ-TRUE**) $\langle i_1 = < i_2 , \sigma \rangle \rightarrow \langle \text{true} , \sigma \rangle$ dacă $i_1 \leq i_2$

$$\frac{\langle a_1 , \sigma \rangle \rightarrow \langle a'_1 , \sigma \rangle}{\langle a_1 = < a_2 , \sigma \rangle \rightarrow \langle a'_1 = < a_2 , \sigma \rangle} \quad \frac{\langle a_2 , \sigma \rangle \rightarrow \langle a'_2 , \sigma \rangle}{\langle a_1 = < a_2 , \sigma \rangle \rightarrow \langle a_1 = < a'_2 , \sigma \rangle}$$

Semantica expresiilor booleene

□ Semantica operatorului de comparație

(LEQ-FALSE) $\langle i_1 = < i_2 , \sigma \rangle \rightarrow \langle \text{false} , \sigma \rangle$ dacă $i_1 > i_2$

(LEQ-TRUE) $\langle i_1 = < i_2 , \sigma \rangle \rightarrow \langle \text{true} , \sigma \rangle$ dacă $i_1 \leq i_2$

$$\frac{\langle a_1 , \sigma \rangle \rightarrow \langle a'_1 , \sigma \rangle}{\langle a_1 = < a_2 , \sigma \rangle \rightarrow \langle a'_1 = < a_2 , \sigma \rangle} \quad \frac{\langle a_2 , \sigma \rangle \rightarrow \langle a'_2 , \sigma \rangle}{\langle a_1 = < a_2 , \sigma \rangle \rightarrow \langle a_1 = < a'_2 , \sigma \rangle}$$

□ Semantica negației

(!-FALSE) $\langle \text{not(true)} , \sigma \rangle \rightarrow \langle \text{false} , \sigma \rangle$

(!-TRUE) $\langle \text{not(false)} , \sigma \rangle \rightarrow \langle \text{true} , \sigma \rangle$

Semantica expresiilor booleene

□ Semantica operatorului de comparație

(LEQ-FALSE) $\langle i_1 = < i_2 , \sigma \rangle \rightarrow \langle \text{false} , \sigma \rangle$ dacă $i_1 > i_2$

(LEQ-TRUE) $\langle i_1 = < i_2 , \sigma \rangle \rightarrow \langle \text{true} , \sigma \rangle$ dacă $i_1 \leq i_2$

$$\frac{\langle a_1 , \sigma \rangle \rightarrow \langle a'_1 , \sigma \rangle}{\langle a_1 = < a_2 , \sigma \rangle \rightarrow \langle a'_1 = < a_2 , \sigma \rangle} \quad \frac{\langle a_2 , \sigma \rangle \rightarrow \langle a'_2 , \sigma \rangle}{\langle a_1 = < a_2 , \sigma \rangle \rightarrow \langle a_1 = < a'_2 , \sigma \rangle}$$

□ Semantica negației

(!-FALSE) $\langle \text{not(true)} , \sigma \rangle \rightarrow \langle \text{false} , \sigma \rangle$

(!-TRUE) $\langle \text{not(false)} , \sigma \rangle \rightarrow \langle \text{true} , \sigma \rangle$

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

Semantica expresiilor booleene

□ Semantica și-ului

(AND-FALSE) $\langle \text{and}(\text{false}, b_2), \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle$

(AND-TRUE) $\langle \text{and}(\text{true}, b_2), \sigma \rangle \rightarrow \langle b_2, \sigma \rangle$

$$\frac{\langle b_1, \sigma \rangle \rightarrow \langle b'_1, \sigma \rangle}{\langle \text{and}(b_1, b_2), \sigma \rangle \rightarrow \langle \text{and}(b'_1, b_2), \sigma \rangle}$$

Semantica compunerii și a blocurilor

□ Semantica blocurilor

(BLOCK) $\langle \{ s \} , \sigma \rangle \rightarrow$

Semantica compunerii și a blocurilor

□ Semantica blocurilor

(BLOCK) $\langle \{ s \} , \sigma \rangle \rightarrow \langle s , \sigma \rangle$

Semantica compunerii și a blocurilor

□ Semantica blocurilor

(BLOCK) $\langle \{ s \} , \sigma \rangle \rightarrow \langle s , \sigma \rangle$

□ Semantica compunerii secvențiale

(NEXT-STATEMENT) $\langle \text{skip} ; s_2 , \sigma \rangle \rightarrow \langle s_2 , \sigma \rangle$

$$\frac{\langle s_1 , \sigma \rangle \rightarrow \langle s'_1 , \sigma' \rangle}{\langle s_1 ; s_2 , \sigma \rangle \rightarrow \langle s'_1 ; s_2 , \sigma' \rangle}$$

Semantica compunerii și a blocurilor

□ Semantica blocurilor

(BLOCK) $\langle \{ s \} , \sigma \rangle \rightarrow \langle s , \sigma \rangle$

□ Semantica compunerii secvențiale

(NEXT-STATEMENT) $\langle \text{skip} ; s_2 , \sigma \rangle \rightarrow \langle s_2 , \sigma \rangle$

$$\frac{\langle s_1 , \sigma \rangle \rightarrow \langle s'_1 , \sigma' \rangle}{\langle s_1 ; s_2 , \sigma \rangle \rightarrow \langle s'_1 ; s_2 , \sigma' \rangle}$$

□ Semantica atribuirii

(ASGN) $\langle x = i , \sigma \rangle \rightarrow \langle \text{skip} , \sigma' \rangle \quad \text{dacă } \sigma' = \sigma_{x \leftarrow i}$

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

Semantica lui if

□ Semantica lui if

(IF-TRUE) $\langle \text{if} (\text{true}, bl_1, bl_2), \sigma \rangle \rightarrow \langle bl_1, \sigma \rangle$

(IF-FALSE) $\langle \text{if} (\text{false}, bl_1, bl_2), \sigma \rangle \rightarrow \langle bl_2, \sigma \rangle$

$$\frac{\langle b, \sigma \rangle \rightarrow \langle b', \sigma \rangle}{\langle \text{if} (b, bl_1, bl_2), \sigma \rangle \rightarrow \langle \text{if} (b', bl_1, bl_2), \sigma \rangle}$$

Semantica lui if

□ Semantica lui if

(IF-TRUE) $\langle \text{if} (\text{true}, bl_1, bl_2), \sigma \rangle \rightarrow \langle bl_1, \sigma \rangle$

(IF-FALSE) $\langle \text{if} (\text{false}, bl_1, bl_2), \sigma \rangle \rightarrow \langle bl_2, \sigma \rangle$

$$\frac{\langle b, \sigma \rangle \rightarrow \langle b', \sigma \rangle}{\langle \text{if} (b, bl_1, bl_2), \sigma \rangle \rightarrow \langle \text{if} (b', bl_1, bl_2), \sigma \rangle}$$

□ Semantica lui while

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

Semantica lui if

□ Semantica lui if

(IF-TRUE) $\langle \text{if} (\text{true}, bl_1, bl_2), \sigma \rangle \rightarrow \langle bl_1, \sigma \rangle$

(IF-FALSE) $\langle \text{if} (\text{false}, bl_1, bl_2), \sigma \rangle \rightarrow \langle bl_2, \sigma \rangle$

$$\frac{\langle b, \sigma \rangle \rightarrow \langle b', \sigma \rangle}{\langle \text{if} (b, bl_1, bl_2), \sigma \rangle \rightarrow \langle \text{if} (b', bl_1, bl_2), \sigma \rangle}$$

□ Semantica lui while

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

□ Semantica programelor

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

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

Semantica small-step a lui IMP

Execuție pas cu pas

$$\langle i = 3 ; \text{while } (0 \leq i , \{ i = i + -4 \}) , \perp \rangle \xrightarrow{P_{GM}}$$

Semantica small-step a lui IMP

Execuție pas cu pas

$$\langle i = 3 ; \text{while } (0 \leq i, \{ i = i + -4 \}) , \perp \rangle \xrightarrow{P_{GM}} \\ \langle \text{while } (0 \leq i, \{ i = i + -4 \}) , i \mapsto 3 \rangle \xrightarrow{\text{WHILE}}$$

Semantica small-step a lui IMP

Execuție pas cu pas

$\langle i = 3 ; \text{while } (0 \leq i, \{ i = i + -4 \}) , \perp \rangle \xrightarrow{P_{GM}}$

$\langle \text{while } (0 \leq i, \{ i = i + -4 \}) , i \mapsto 3 \rangle \xrightarrow{\text{WHILE}}$

$\langle \text{if } (0 \leq i, i = i + -4) ; \text{while } (0 \leq i, \{ i = i + -4 \}) , \text{skip}) , i \mapsto 3 \rangle \xrightarrow{I_D}$

Semantica small-step a lui IMP

Execuție pas cu pas

$\langle i = 3 ; \text{while } (0 \leq i, \{ i = i + -4 \}) , \perp \rangle \xrightarrow{P_{GM}}$

$\langle \text{while } (0 \leq i, \{ i = i + -4 \}) , i \mapsto 3 \rangle \xrightarrow{\text{WHILE}}$

$\langle \text{if } (0 \leq i, i = i + -4 ; \text{while } (0 \leq i, \{ i = i + -4 \}) , \text{skip}) , i \mapsto 3 \rangle \xrightarrow{\text{Id}}$

$\langle \text{if } (0 \leq 3, i = i + -4 ; \text{while } (0 \leq i, \{ i = i + -4 \}) , \text{skip}) , i \mapsto 3 \rangle \xrightarrow{\text{LEQ-TRUE}}$

Semantica small-step a lui IMP

Execuție pas cu pas

$\langle i = 3 ; \text{while } (0 \leq i, \{ i = i + -4 \}) , \perp \rangle \xrightarrow{P_{GM}}$

$\langle \text{while } (0 \leq i, \{ i = i + -4 \}) , i \mapsto 3 \rangle \xrightarrow{\text{WHILE}}$

$\langle \text{if } (0 \leq i, i = i + -4 ; \text{while } (0 \leq i, \{ i = i + -4 \}) , \text{skip}) , i \mapsto 3 \rangle \xrightarrow{\text{ID}}$

$\langle \text{if } (0 \leq 3, i = i + -4 ; \text{while } (0 \leq i, \{ i = i + -4 \}) , \text{skip}) , i \mapsto 3 \rangle \xrightarrow{\text{LEQ-TRUE}}$

$\langle \text{if } (\text{true}, i = i + -4 ; \text{while } (0 \leq i, \{ i = i + -4 \}) , \text{skip}) , i \mapsto 3 \rangle \xrightarrow{\text{IF-TRUE}}$

Semantica small-step a lui IMP

Execuție pas cu pas

$\langle i = 3 ; \text{while } (0 \leq i, \{ i = i + -4 \}) , \perp \rangle \xrightarrow{P_{GM}}$

$\langle \text{while } (0 \leq i, \{ i = i + -4 \}) , i \mapsto 3 \rangle \xrightarrow{\text{WHILE}}$

$\langle \text{if } (0 \leq i, i = i + -4 ; \text{while } (0 \leq i, \{ i = i + -4 \}) , \text{skip}) , i \mapsto 3 \rangle \xrightarrow{\text{ID}}$

$\langle \text{if } (0 \leq 3, i = i + -4 ; \text{while } (0 \leq i, \{ i = i + -4 \}) , \text{skip}) , i \mapsto 3 \rangle \xrightarrow{\text{LEQ-TRUE}}$

$\langle \text{if } (\text{true}, i = i + -4 ; \text{while } (0 \leq i, \{ i = i + -4 \}) , \text{skip}) , i \mapsto 3 \rangle \xrightarrow{\text{IF-TRUE}}$

$\langle i = i + -4 ; \text{while } (0 \leq i, \{ i = i + -4 \}) , i \mapsto 3 \rangle \xrightarrow{\text{ID}}$

...

Avantaje și dezavantaje

Semantica operațională

- + Definește precis noțiunea de pas computațional
- + Semnalează erorile, oprind execuția
- + Execuția devine ușor de urmărit și depanat
- Regulile structurale sunt evidente și deci plăcătitor de scris
- Nemodular: adăugarea unei trăsături noi poate solicita schimbarea întregii definiții



Pe săptămâna viitoare!

Curs 10

2021-2022

Fundamentele limbajelor de programare

Cuprins

- 1 O implementare a limbajului IMP în Prolog
- 2 O implementare a semanticii small-step
- 3 Semantica axiomatică
- 4 Semantica denotațională (optional)

O implementare a limbajului IMP în Prolog

Sintaxa BNF a limbajului IMP

$E ::= n \mid x$
| $\text{++}(E)$
| $E + E \mid E - E \mid E * E$

$B ::= \text{true} \mid \text{false}$
| $E <= E \mid E >= E \mid E == E$
| $\text{not}(B) \mid \text{and}(B, B) \mid \text{or}(B, B)$

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

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

Decizii de implementare

- {} si ; sunt operatori

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

- definim un predicat pentru fiecare categorie sintactică

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

- while, if, and, etc sunt functori în Prolog

while(true, skip) este un termen compus

- , , are semnificația obisnuită

- pentru valori numerice folosim întregii din Prolog

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

- pentru identificatori folosim atomii din Prolog

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

Expresiile aritmetice

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

Prolog

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

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

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

```
aexp(A1 + A2) :- aexp(A1), aexp(A2).
```

```
...
```

Expresiile aritmetice

Exemplu

?- aexp(1000).

true.

?- aexp(id).

true.

?- aexp(id + 1000).

true.

?- aexp(2 + 1000).

true.

?- aexp(x * y).

true.

?- aexp(- x).

false.

Expresiile booleene

```
B ::= true | false  
| E <= E | E >= E | E == E  
| not(B) | and(B , B) | or(B , B)
```

Prolog

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

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

```
...
```

Expresiile booleene

Exemplu

?- bexp(true).

true.

?- bexp(id).

false.

?- bexp(not(1 = \leq 2)).

true.

?- bexp(or(1 = \leq 2,true)).

true.

?- bexp(or(a = \leq b,true)).

true.

?- bexp(not(a)).

false.

?- bexp(!(a)).

false.

Instructiunile

```
C ::= skip
  | x = E
  | if(B, C, C)
  | while(B, C)
  | {C} | C; C
```

Prolog

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

Instructiunile

Exemplu

?- stmt(id = 5).

true.

?- stmt(id = a).

true.

?- stmt(3 = 6).

false.

?- stmt(if(true, x=2;y=3, x=1;y=0)).

true.

?- stmt(while(x =< 0,skip)).

true.

?- stmt(while(x =< 0,)).

false.

?- stmt(while(x =< 0,skip)).

true .

Programele

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

Prolog

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

Exemplu

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

```
?- test0.  
true.
```

O implementare a semanticii small-step

Semantica small-step

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

Semantica small-step

- Defineste cel mai mic pas de executie ca o relatie de tranzitie intre configuratii:

$\langle cod, \sigma \rangle \rightarrow \langle cod', \sigma' \rangle$ **smallstep(Cod, S1, Cod', S2)**

- Executia se obtine ca o succesiune de astfel de tranzitii.
- Starea executiei unui program IMP la un moment dat este o functie partială: $\sigma = n \mapsto 10, sum \mapsto 0$, etc.

Reprezentarea stărilor în Prolog

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

Semantica expresiilor aritmetice

□ Semantica unei variabile

$\langle x, \sigma \rangle \rightarrow \langle i, \sigma \rangle \text{ dacă } i = \sigma(x)$

Prolog

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

Semantica expresiilor aritmetice

□ Semantica adunării a două expresii aritmetice

$\langle i_1 + i_2 , \sigma \rangle \rightarrow \langle i , \sigma \rangle \quad \text{dacă } i = i_1 + i_2$

$$\frac{\langle a_1 , \sigma \rangle \rightarrow \langle a'_1 , \sigma \rangle}{\langle a_1 + a_2 , \sigma \rangle \rightarrow \langle a'_1 + a_2 , \sigma \rangle} \qquad \frac{\langle a_2 , \sigma \rangle \rightarrow \langle a'_2 , \sigma \rangle}{\langle a_1 + a_2 , \sigma \rangle \rightarrow \langle a_1 + a'_2 , \sigma \rangle}$$

□ Ordine nespecificată de evaluare a argumentelor

Prolog

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

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

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

Semantica expresiilor aritmetice

Exemplu

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

AE = 1+b,

S = [vi(a, 1), vi(b, 2)] .

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

AE = 1+2,

S = [vi(a, 1), vi(b, 2)] .

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

AE = 3,

S = [vi(a, 1), vi(b, 2)]

Semantica expresiilor aritmetice

Exemplu

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

AE = $1+b$,

S = [vi(a, 1), vi(b, 2)] .

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

AE = $1+2$,

S = [vi(a, 1), vi(b, 2)] .

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

AE = 3,

S = [vi(a, 1), vi(b, 2)]

- Semantica * si - se definesc similar.

Semantica expresiilor booleene

□ Semantica operatorului de comparatie

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

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

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle}{\langle a_1 = < a_2, \sigma \rangle \rightarrow \langle a'_1 = < a_2, \sigma \rangle} \quad \frac{\langle a_2, \sigma \rangle \rightarrow \langle a'_2, \sigma \rangle}{\langle a_1 = < a_2, \sigma \rangle \rightarrow \langle a_1 = < a'_2, \sigma \rangle}$$

Prolog

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

Semantica expresiilor Booleene

□ Semantica negatiei

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

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

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

Prolog

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

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

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

Semantica compunerii si a blocurilor

- Semantica blocurilor

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

- Semantica compunerii secentiale

$$\langle \{ \} ; s_2 , \sigma \rangle \rightarrow \langle s_2 , \sigma \rangle \quad \frac{\langle s_1 , \sigma \rangle \rightarrow \langle s'_1 , \sigma' \rangle}{\langle s_1 ; s_2 , \sigma \rangle \rightarrow \langle s'_1 ; s_2 , \sigma' \rangle}$$

Prolog

```
smallstepS([E], S, E, S).
```

```
smallstepS((skip;St2), S, St2, S).
```

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

Semantica atribuirii

□ Semantica atribuirii

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

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

Prolog

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

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

Semantica lui if

□ Semantica lui if

$$\langle \text{if}(\text{true}, bl_1, bl_2), \sigma \rangle \rightarrow \langle bl_1, \sigma \rangle$$

$$\langle \text{if}(\text{false}, bl_1, bl_2), \sigma \rangle \rightarrow \langle bl_2, \sigma \rangle$$

$$\frac{\langle b, \sigma \rangle \rightarrow \langle b', \sigma \rangle}{\langle \text{if}(b, bl_1, bl_2), \sigma \rangle \rightarrow \langle \text{if}(b', bl_1, bl_2), \sigma \rangle}$$

Prolog

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

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

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

Semantica lui while

□ Semantica lui while

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

Prolog

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

Semantica programelor

□ Semantica programelor

$$\frac{\langle a_1, \sigma_1 \rangle \rightarrow \langle a_2, \sigma_2 \rangle}{\langle (\text{skip}, a_1), \sigma_1 \rangle \rightarrow \langle (\text{skip}, a_2), \sigma_2 \rangle}$$
$$\frac{\langle s_1, \sigma_1 \rangle \rightarrow \langle s_2, \sigma_2 \rangle}{\langle (s_1, a), \sigma_1 \rangle \rightarrow \langle (s_2, a), \sigma_2 \rangle}$$

Prolog

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

Executia programelor

Prolog

```
run(skip,I,_,I) :- integer(I).  
run(St1,AE1,S1,I) :- smallstepP(St1,AE1,S1,St2,AE2,S2),  
                      run(St2,AE2,S2,I).  
  
run_program(Name) :- defpg(Name,{P},E), run(P,E, [],I),  
                   write(I).
```

Exemplu

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

```
?- run_program(pg2).
```

55

true

Executia programelor: trace

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

Executia programelor: trace

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

Prolog

```
mytrace(skip,I,_) :- integer(I).  
mytrace(St1,AE1,S1) :- smallstepP(St1,AE1,S1,St2,AE2,S2),  
                      write(St2),nl,  
                      write(AE2),nl,  
                      write(S2),nl,  
                      mytrace(St2,AE2,S2).  
  
trace_program(Name) :- defpg(Name,{P},E),  
                     mytrace(P,E,[]).
```

Executia programelor: trace_program

Exemplu

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

Semantica axiomatică

Semantica Axiomatică

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

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

unde:

- S este o instrucțiune (Stmt)
- Pre (precondiție), respectiv $Post$ (postcondiție) sunt asertiiuni logice asupra stării sistemului înaintea, respectiv după execuția lui S
- Limbajul asertiiunilor este un limbaj de ordinul I.

Semantica Axiomatică

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

- dacă programul se execută dintr-o stare inițială care satisfacă *Pre*
- și execuția se termină
- atunci se ajunge într-o stare finală care satisfacă *Post*.

Putem verifica corectitudine parțială.

Semantica Axiomatică

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

- dacă programul se execută dintr-o stare inițială care satisfacă *Pre*
- și execuția se termină
- atunci se ajunge într-o stare finală care satisfacă *Post*.

Putem verifica corectitudine parțială.

Exemplu

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

Semantica Axiomatică

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

- dacă programul se execută dintr-o stare inițială care satisfacă *Pre*
- și execuția se termină
- atunci se ajunge într-o stare finală care satisfacă *Post*.

Putem verifica corectitudine parțială.

Exemplu

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

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

Sistem de reguli pentru logica Floyd-Hoare

$$(\rightarrow) \quad \frac{P_1 \rightarrow P_2 \quad \{P_2\} c \{Q_2\} \quad Q_2 \rightarrow Q_1}{\{P_1\} c \{Q_1\}}$$

$$(\vee) \quad \frac{\{P_1\} c \{Q\} \quad \{P_2\} c \{Q\}}{\{P_1 \vee P_2\} c \{Q\}}$$

$$(\wedge) \quad \frac{\{P\} c \{Q_1\} \quad \{P\} c \{Q_2\}}{\{P\} c \{Q_1 \wedge Q_2\}}$$

Logica Floyd-Hoare pentru IMP1

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

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

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

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

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

Logica Floyd-Hoare pentru IMP1

- regula pentru atribuire

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

Exemplu

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

Logica Floyd-Hoare pentru IMP1

- regula pentru atribuire

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

Exemplu

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

- regula pentru condiții

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

Exemplu

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

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

Invarianți pentru while

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

- Se determină un invariant I și se folosește următoarea regulă:

$$\text{(INV)} \quad \frac{P \rightarrow I \quad \{b \wedge I\} c \{I\} \quad (I \wedge \neg b) \rightarrow Q}{\{P\} \text{while}(b) c \{Q\}}$$

Invarianți pentru while

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

- Se determină un invariant I și se folosește următoarea regulă:

$$\text{(INV)} \quad \frac{P \rightarrow I \quad \{b \wedge I\} c \{I\} \quad (I \wedge \neg b) \rightarrow Q}{\{P\} \text{while } (b) c \{Q\}}$$

Invariantul trebuie să satisfacă următoarele proprietăți:

- să fie adevărat inițial
- să rămână adevărat după executarea unui ciclu
- să implice postcondiția la ieșirea din buclă

Invarianți pentru while

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

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

$\{y = n!\}$

Invarianți pentru while

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

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

$\{y = n!\}$

- Invariantul / este $y = x!$

Invarianți pentru while

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

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

$\{y = n!\}$

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

Dafny

- Dezvoltat la Microsoft Research
- Un limbaj imperativ compilat open-source
- Suportă demonstrații formale folosind **precondiții**, **postcondiții**, **invarianți de bucle**
- Demonstrează și terminarea programelor
- Concepte din diferite paradigmă de programare
 - Programare imperativă: `if`, `while`, `:=`, ...
 - Programare funcțională: `function`, `datatype`, ...
 - ...

- Pagina limbajului Dafny

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

- Pagina de Github

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

- Dafny in browser

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

- Tutorial

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

Dafny - Hello World

Dafny - fără erori

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

Dafny - Hello World

Dafny - fără erori

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

Dafny - erori

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

Dafny - maximul a două numere

Următoarea metodă calculează maximul a doi întregi:

Dafny

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

Dar cum verificăm formal acest lucru?

Dafny - maximul a două numere

Adăugăm postcondiții!

Dafny

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

Dafny - maximul a două numere

Adăugăm postcondiții!

Dafny

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

Dar este de ajuns condiția de mai sus?

Dafny - maximul a două numere

Adăugăm postcondiții!

Dafny

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

Dar este de ajuns condiția de mai sus?

O metodă este reprezentată prin precondițiile și postcondițiile pe care le satisfacă, iar codul este "ignorat" ulterior.

În exemplul de mai sus, pentru $x = 3$ și $y = 6$, $z = 7$ satisfacă postcondiția (dacă ignorăm codul) dar nu este maximul dintre x și y .

Dafny - maximul a două numere

Dafny

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

Dafny - suma primelor n numere impare

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

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

Dafny - suma primelor n numere impare

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

Dafny

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

Dafny - suma primelor n numere impare

Adăugăm postcondiția!

Dafny

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

Dar Dafny nu reușește să o demonstreze!

Dafny - suma primelor n numere impare

Adăugăm postcondiția!

Dafny

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

Dar Dafny nu reușește să o demonstreze! Trebuie să îi spunem ce se întâmplă în buclă prin invariante.

Dafny - suma primelor n numere impare

Dafny

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

Dafny - suma primelor n numere impare

Dafny

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

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

Dafny - suma primelor n numere impare

Dafny

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

Dafny - suma primelor n numere impare

Dafny

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

Totuși Dafny nu reușește să demonstreze postcondiția! De ce?

Dafny - suma primelor n numere impare

Dafny

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

Totuși Dafny nu reușește să demonstreze postcondiția! De ce?
Ce se întâmplă dacă $n < 0$?

Dafny - suma primelor n numere impare

Dafny

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

Dafny - funcția factorial

Dafny

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

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

Semantica denotațională (optional)

Semantica denotațională

- Introdusă de Christopher Strachey și Dana Scott (1970)
- Semantica operațională, ca un interpretor, descrie **cum** să evaluăm un program.
- **Semantica denotațională**, ca un compilator, descrie o traducere a limbajului într-un limbaj diferit cu semantică cunoscută, anume matematică.
- Semantica denotațională definește ce înseamnă un program ca o funcție matematică.

Semantica denotațională

- Definim stările memoriei ca fiind funcții parțiale de la mulțimea identificatorilor la mulțimea valorilor:

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

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

Semantica denotațională

- Definim stările memoriei ca fiind funcții parțiale de la mulțimea identificatorilor la mulțimea valorilor:

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

- Asociem fiecărei categorii sintactice o categorie semantică.
- Fiecare construcție sintactică va avea o denotație (interpretare) în categoria semantică respectivă. De exemplu:
 - denotația unei expresii aritmetice este o funcție parțială de la mulțimea stărilor memoriei la mulțimea valorilor (\mathbb{Z}):
$$[[__]] : AExp \rightarrow (State \rightarrow \mathbb{Z})$$
 - denotația unei instrucțiuni este o funcție parțială de la mulțimea stărilor memoriei la mulțimea stărilor memoriei:
$$[[__]] : Stmt \rightarrow (State \rightarrow State)$$

Semantica denotațională

$\text{State} = \text{Id} \rightarrow \mathbb{Z}$

$[\![__]\!] : AExp \rightarrow (\text{State} \rightarrow \mathbb{Z})$

$[\![__]\!] : Stmt \rightarrow (\text{State} \rightarrow \text{State})$

Atribuirea: $x = \text{expr}$

- Asociem expresiilor aritmetice funcții de la starea memoriei la valori:

- Asociem instrucțiunilor funcții de la starea memoriei la starea (următoare) a memoriei.

Semantica denotațională

$\text{State} = \text{Id} \rightarrow \mathbb{Z}$

$[[__]] : AExp \rightarrow (\text{State} \rightarrow \mathbb{Z})$

$[[__]] : Stmt \rightarrow (\text{State} \rightarrow \text{State})$

Atribuirea: $x = \text{expr}$

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

Semantica denotațională: expresii

$$\text{State} = \text{Id} \rightarrow \mathbb{Z}$$

□ Domenii semantice:

$$[[__]] : AExp \rightarrow (\text{State} \rightarrow \mathbb{Z})$$

$$[[__]] : BExp \rightarrow (\text{State} \rightarrow \{T, F\})$$

$$[[__]] : Stmt \rightarrow (\text{State} \rightarrow \text{State})$$

Semantica denotațională: expresii

$$\text{State} = \text{Id} \rightarrow \mathbb{Z}$$

- Domenii semantice:

$$[[__]] : AExp \rightarrow (\text{State} \rightarrow \mathbb{Z})$$

$$[[__]] : BExp \rightarrow (\text{State} \rightarrow \{T, F\})$$

$$[[__]] : Stmt \rightarrow (\text{State} \rightarrow \text{State})$$

- Semantica denotațională este compozitională:

- semantica expresiilor aritmetice

$$[[n]](s) = n$$

$$[[x]](s) = s(x)$$

$$[[e1 + e2]](s) = [[e1]](s) + [[e2]](s)$$

Semantica denotațională: expresii

$$\text{State} = \text{Id} \rightarrow \mathbb{Z}$$

- Domenii semantice:

$$[[__]] : AExp \rightarrow (\text{State} \rightarrow \mathbb{Z})$$

$$[[__]] : BExp \rightarrow (\text{State} \rightarrow \{T, F\})$$

$$[[__]] : Stmt \rightarrow (\text{State} \rightarrow \text{State})$$

- Semantica denotațională este compozitională:

- semantica expresiilor aritmetice

$$[[n]](s) = n$$

$$[[x]](s) = s(x)$$

$$[[e1 + e2]](s) = [[e1]](s) + [[e2]](s)$$

- semantica expresiilor booleene

$$[[\text{true}]](s) = T, [[\text{false}]](s) = F$$

$$[[\neg b]](s) = \neg b$$

$$[[e1 <= e2]](s) = [[e1]](s) <= [[e2]](s)$$

Semantica denotațională: instrucțiuni

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

□ Domenii semantice:

$$[[__]] : AExp \rightarrow (State \rightarrow \mathbb{Z})$$

$$[[__]] : BExp \rightarrow (State \rightarrow \{T, F\})$$

$$[[__]] : Stmt \rightarrow (State \rightarrow State)$$

Semantica denotațională: instrucțiuni

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

- Domenii semantice:

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

$$[[_]] : BExp \rightarrow (State \rightarrow \{T, F\})$$

$$[[_]] : Stmt \rightarrow (State \rightarrow State)$$

- Semantica instrucțiunilor:

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

$$[[c_1; c_2]] = [[c_2]] \circ [[c_1]]$$

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

Semantica denotațională: instrucțiuni

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

□ Domenii semantice:

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

$$[[_]] : BExp \rightarrow (State \rightarrow \{T, F\})$$

$$[[_]] : Stmt \rightarrow (State \rightarrow State)$$

□ Semantica instrucțiunilor:

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

$$[[c_1; c_2]] = [[c_2]] \circ [[c_1]]$$

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

$$[[\text{if } (b) \text{ c}_1 \text{ else c}_2]](s) = \begin{cases} [[c_1]](s), & \text{dacă } [[b]](s) = T \\ [[c_2]](s), & \text{dacă } [[b]](s) = F \end{cases}$$

Semantica denotațională

Exemplu

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

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

Semantica denotațională

Exemplu

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

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

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

Semantica denotațională

Exemplu

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

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

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

Cum definim semantica denotațională pentru `while`?

Mulțimea funcțiilor parțiale

Fie X și Y două mulțimi.

- $Pfn(X, Y)$ mulțimea funcțiilor parțiale de la X la Y , adică
 $Pfn(X, Y) = X \rightharpoonup Y$
- Pentru $f \in Pfn(X, Y)$ notăm cu $dom(f)$ mulțimea elementelor din X pentru care funcția este definită.
Atunci $dom(f) \subseteq X$ și $f|_{dom(f)} : dom(f) \rightarrow Y$ este funcție.

Mulțimea funcțiilor parțiale

Fie X și Y două mulțimi.

- $Pfn(X, Y)$ mulțimea funcțiilor parțiale de la X la Y , adică
 $Pfn(X, Y) = X \rightharpoonup Y$
- Pentru $f \in Pfn(X, Y)$ notăm cu $dom(f)$ mulțimea elementelor din X pentru care funcția este definită.
Atunci $dom(f) \subseteq X$ și $f|_{dom(f)} : dom(f) \rightarrow Y$ este funcție.
- Fie $\perp : X \rightharpoonup Y$ unica funcție cu $dom(\perp) = \emptyset$ (funcția care nu este definită în nici un punct).
- Definim pe $Pfn(X, Y)$ următoarea relație:

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

Mulțimea funcțiilor parțiale

Fie X și Y două mulțimi.

- $Pfn(X, Y)$ mulțimea funcțiilor parțiale de la X la Y , adică
 $Pfn(X, Y) = X \rightharpoonup Y$
- Pentru $f \in Pfn(X, Y)$ notăm cu $dom(f)$ mulțimea elementelor din X pentru care funcția este definită.
Atunci $dom(f) \subseteq X$ și $f|_{dom(f)} : dom(f) \rightarrow Y$ este funcție.
- Fie $\perp : X \rightharpoonup Y$ unica funcție cu $dom(\perp) = \emptyset$ (funcția care nu este definită în nici un punct).
- Definim pe $Pfn(X, Y)$ următoarea relație:

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

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

(mulțime parțial ordonată completă în care \perp este cel mai mic element)

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

Exemplu

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

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

- F este o funcție continuă,

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

Exemplu

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

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

- F este o funcție continuă, deci putem aplica
- Teorema Knaster-Tarski
Fie $g_n = F^n(\perp)$ și $f = \bigvee_n g_n$.
Stim că f este cel mai mic punct fix al funcției F , deci $F(f) = f$.

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

Exemplu

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

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

- F este o funcție continuă, deci putem aplica
- Teorema Knaster-Tarski
Fie $g_n = F^n(\perp)$ și $f = \bigvee_n g_n$.
Stim că f este cel mai mic punct fix al funcției F , deci $F(f) = f$.
- Demonstrăm prin inducție după n că:
 $\text{dom}(g_n) = \{0, \dots, n\}$ și $g_n(k) = k!$ oricare $k \in \text{dom}(g_n)$
- $f : \mathbb{N} \rightarrow \mathbb{N}$ este funcția factorial.

Semantica denotațională pentru while

`while (b) c`

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

Semantica denotațională pentru while

while (b) c

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

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

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

Semantica denotațională pentru while

`while (b) c`

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

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

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

$[[_]] : Stmt \rightarrow (State \rightarrow State)$

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

Avantaje și dezavantaje

Semantica operațională

- + Definește precis noțiunea de pas computațional
- + Semnalează erorile, oprind execuția
- + Execuția devine ușor de urmărit și depanat
- Regulile structurale sunt evidente și deci plăcătitor de scris
- Nemodular: adăugarea unei trăsături noi poate solicita schimbarea întregii definiții

Semantica denotațională

- + Formală, matematică, foarte precisă
- + Compozițională (morfisme și compuneri de funcții)
- Domeniile devin din ce în ce mai complexe.



Pe săptămâna viitoare!

Curs 11

2021-2022

Fundamentele limbajelor de programare

Cuprins

1 Implementarea Mini-Haskell în Haskell

2 Monade în Haskell

Implementarea Mini-Haskell în Haskell

Mini-Haskell

Vom defini folosind Haskell un mini limbaj funcțional și un interpreter pentru acesta.

Mini-Haskell

Vom defini folosind Haskell un mini limbaj funcțional și un interpretor pentru acesta.

- Limbajul Mini-Haskell conține:
 - expresii de tip Int
 - expresii de tip funcție (λ -expresii)
 - expresii provenite din aplicarea functiilor

Mini-Haskell

Vom defini folosind Haskell un mini limbaj funcțional și un interpretor pentru acesta.

- Limbajul Mini-Haskell conține:
 - expresii de tip Int
 - expresii de tip funcție (λ -expresii)
 - expresii provenite din aplicarea funcțiilor
- Pentru a defini semantica limbajului vom introduce domeniile semantice (valorile) asociate expresiilor limbajului.
- Pentru a evalua (interpreta) expresiile vom defini un mediu de evaluare în care vom retine variabilele și valorile curente asociate.

Mini-Haskell (λ -calcul cu întregi). Sintaxă

```
type Name = String

data Term = Var Name
          | Con Integer
          | Term :+: Term
          | Lam Name Term
          | App Term Term
deriving (Show)
```

Program - Exemplu

λ -expresia $(\lambda x.x + x)(10 + 11)$

este definită astfel:

pgm :: Term

pgm = App

```
(Lam "x" ((Var "x") :+: (Var "x")))
  ((Con 10) :+: (Con 11))
```

Program - Exemplu

```
pgm :: Term
pgm = App
  (Lam "y"
    (App
      (App
        (Lam "f"
          (Lam "y"
            (App (Var "f") (Var "y")))
          )
        )
      (Lam "x"
        (Var "x" :+: Var "y")
        )
      )
    (Con 3)
    )
  )
(Con 4)
```

Ce λ -expresie defineste termenul de mai sus?

Domenii

Domeniul valorilor

```
data Value = Num Integer  
          | Fun (Value -> Value)  
          | Wrong -- pentru reprezentarea erorilor
```

Mediul de evaluare

```
type Environment = [(Name, Value)]
```

Domeniul de evaluare

Fiecarei expresii i se va asocia ca denotație o funcție de la medii de evaluare la valori:

```
interp :: Term -> Environment -> Value
```

Afișarea expresiilor

```
instance Show Value where
    show (Num x) = show x
    show (Fun _) = "<function>"
    show Wrong   = "<wrong>"
```

Observație

Funcțiile nu pot fi afișate ca atare, ci doar generic.

Evaluarea expresiilor Mini-Haskell în Haskell

```
interp :: Term -> Environment -> Value
interp (Con i) _ = Num i
interp (t1 :+: t2) env = add (interp t1 env) (interp t2 env)
add :: Value -> Value -> Value
add (Num i) (Num j) = Num $ i + j
add _ _ = Wrong
```

Evaluarea expresiilor Mini-Haskell în Haskell

```
interp :: Term -> Environment -> Value
interp (Var x) env = lookupM x env

lookupM :: Name -> Environment -> Value
lookupM x env = case lookup x env of
  Just v -> v
  Nothing -> Wrong

-- lookup din modulul Data.List
lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup key [] = Nothing
lookup key ((x,y) : xys)
  | key == x = Just y
  | otherwise = lookup key xys
```

Evaluarea expresiilor Mini-Haskell în Haskell

```
interp :: Term -> Environment -> Value
```

```
interp (Lam x e) env = Fun $ \ v -> interp e ((x,v):env)
```

Evaluarea expresiilor Mini-Haskell în Haskell

```
interp :: Term -> Environment -> Value
interp (Lam x e) env = Fun $ \ v -> interp e ((x,v):env)
interp (App t1 t2) env = apply f v
  where
    f = interp t1 env
    v = interp t2 env
apply :: Value -> Value -> Value
apply (Fun k) v = k v
apply _ _ = Wrong
```

Implementarea Mini-Haskell în Haskell

```
interp :: Term -> Environment -> Value
```

```
interp (Var x) env = lookupM x env
  where lookupM x env = case lookup x env of
      Just v -> v
      Nothing -> Wrong
```

```
interp (Con i) _ = Num i
```

```
interp (t1 :+: t2) env = add (interp t1 env) (interp t2 env)
  where add (Num i) (Num j) = Num $ i + j
        add _ _ = Wrong
```

```
interp (Lam x e) env = Fun $ \ v -> interp e ((x,v):env)
```

```
interp (App t1 t2) env = apply (interp t1 env) (interp t2 env)
  where apply (Fun k) v = k v
        apply _ _ = Wrong
```

Evaluarea expresiilor Mini-Haskell în Haskell

```
pgm :: Term
pgm = App
        (Lam "x" ((Var "x") :+: (Var "x")))
        ((Con 10) :+: (Con 11))
```

```
test :: Term -> String
test t = show $ interp t []
```

```
*Main> test pgm
"42"
```

Evaluarea expresiilor Mini-Haskell în Haskell

```
pgmW :: Term
pgmW = App
    (((Var "x") :+: (Var "x")))
    ((Con 10) :+: (Con 11))
```

```
test :: Term -> String
test t = show $ interp t []
```

```
*Main> test pgmW
"<wrong>"
```

Evaluarea expresiilor Mini-Haskell în Haskell

```
pgm :: Term
pgm = App
    (Lam "x" ((Var "x") :+: (Var "x")))
    ((Con 10) :+: (Con 11))

testIO :: Term -> IO ()
testIO t = putStrLn . show $ interp t []

> test pgm
42
```

Evaluarea expresiilor Mini-Haskell în Haskell

```
pgm :: Term
pgm = App
    (Lam "x" ((Var "x") :+: (Var "x")))
    ((Con 10) :+: (Con 11))

testIO :: Term -> IO ()
testIO t = putStrLn . show $ interp t []

> test pgm
42
```

Ce este IO?

Monade în Haskell

Haskell

- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții.
- Funcțiile sunt **pure**: aceleasi rezultate pentru aceleasi intrări.
- Puritatea asigură coherență:
 - O bucată de cod nu poate corupe datele altor bucate de cod.
 - Mai ușor de testat decât codul care interacționează cu mediul.

Haskell

- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții.
- Funcțiile sunt **pure**: aceleasi rezultate pentru aceleasi intrări.
- Puritatea asigură coherență:
 - O bucată de cod nu poate corupe datele altor bucate de cod.
 - Mai ușor de testat decât codul care interacționează cu mediul.

Cum interacționăm cu mediul extern, păstrând puritatea?

Haskell

- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții.
- Funcțiile sunt **pure**: aceleasi rezultate pentru aceleasi intrări.
- Puritatea asigură coherență:
 - O bucătă de cod nu poate corupe datele altor bucăți de cod.
 - Mai ușor de testat decât codul care interacționează cu mediul.

Cum interacționăm cu mediul extern, păstrând puritatea?
Folosim *monade*!

Ce este o monadă?

Există multe răspunsuri, variind între

- O monadă este un **burrito**. <https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/>

Monad →



<https://twitter.com/monadburritos>

Ce este o monadă?

Există multe răspunsuri, variind între

- O monadă este un **burrito**. <https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/>



- "All told, a monad in X is just a monoid in the category of endofunctors in X , with product \times replaced by composition of endofunctors and unit set by the identity endofunctor."

Saunders Mac Lane, Categories for the Working Mathematician, 1998.

Funcții îmbogățite și efecte

- Funcție simplă: $x \mapsto y$
știind x , obținem **direct** y

Funcții îmbogățite și efecte

- Funcție simplă: $x \mapsto y$
știind x , obținem **direct** y

- Funcție îmbogățită: $x \mapsto$



știind x , putem să **extragem** y și producem un **efect**

<https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/>

<https://bartoszmilewski.com/2016/11/30/monads-and-effects/>

Funcții îmbogățite și efecte

Funcție îmbogățită: $x \mapsto$



Exemplu

- Folosind tipul Maybe a

```
data Maybe a = Nothing | Just a
f :: Int -> Maybe Int
f x = if x < 0 then Nothing else (Just x)
```

Funcții îmbogățite și efecte

Funcție îmbogățită: $x \mapsto$



Exemplu

- Folosind un tip care are ca efect un mesaj

```
newtype Writer log a = Writer {runWriter :: (a, log)}
```

```
f :: Int -> Writer String Int
```

```
f x = if x < 0 then (Writer (-x, "negativ"))
      else (Writer (x, "pozitiv"))
```

Logging în Haskell

„Îmbogățim” rezultatul funcției cu mesajul de log.

```
newtype Writer log a = Writer {runWriter :: (a, log)}
```

Observații

- datele de tip Writer log a sunt definite folosind înregistrări
- o dată de tip Writer log a are una din formele
Writer (va,vlog) sau Writer {runWriter = (va,vlog)}
unde va :: a și vlog :: log
- runWriter este funcția proiecție:
runWriter :: Writer log a -> (a, log)
de exemplu runWriter (Writer (1,"msg")) = (1,"msg")

Componerea funcțiilor

- Principala operație pe care o facem cu funcții este **componerea**

$f :: a \rightarrow b$, $g :: b \rightarrow c$, $g . f :: a \rightarrow c$
 $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

Valoarea de tip b este transmisă **direct** funcției g .

Componerea funcțiilor

- Principala operație pe care o facem cu funcții este **componerea**

$f :: a \rightarrow b$, $g :: b \rightarrow c$, $g . f :: a \rightarrow c$
 $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

Valoarea de tip b este transmisă **direct** funcției g .

- Ce facem dacă

$f :: a \rightarrow m\ b$, $g :: b \rightarrow m\ c$

unde m este un **constructor de tip** care îmbogățește tipul?

Componerea funcțiilor

- Principala operație pe care o facem cu funcții este **componerea**

$f :: a \rightarrow b$, $g :: b \rightarrow c$, $g . f :: a \rightarrow c$
 $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

Valoarea de tip b este transmisă **direct** funcției g .

- Ce facem dacă

$f :: a \rightarrow m\ b$, $g :: b \rightarrow m\ c$

unde m este un **constructor de tip** care îmbogățește tipul?

De exemplu,

- $m = \text{Maybe}$
- $m = \text{Writer log}$

- **Atenție!** m trebuie să aibă un singur argument.

Componerea funcțiilor

$f :: a \rightarrow m b$, $g :: b \rightarrow m c$

unde m este un **constructor de tip** care îmbogățește tipul.

Vrem să definim o "componere" pentru funcții îmbogățite

$(\langle=\rangle) :: (b \rightarrow m c) \rightarrow (a \rightarrow m b) \rightarrow a \rightarrow m c$

Atunci când definim $g \langle=\rangle f$ trebuie să **extragem** valoarea întoarsă de f și să o trimitem lui g .

Exemplu: logging în Haskell

„îmbogățim” rezultatul funcției cu mesajul de log.

```
newtype Writer log a = Writer { runWriter :: (a, log) }

logInc :: Int -> Writer String Int
logInc x =
    Writer (x + 1, "Called inc with arg " ++ show x ++ "\n")
```

Problema: Cum calculăm logInc (logInc x)?

Exemplu: logging în Haskell

```
newtype Writer log a = Writer { runWriter :: (a, log) }

logInc :: Int -> Writer String Int
logInc x =
    Writer (x + 1, "Called inc with arg " ++ show x ++ "\n")

logInc2 :: Int -> Writer String Int
logInc2 x = let (y, log1) = runWriter (logInc x)
            (z, log2) = runWriter (logInc y)
            in Writer (z, log1 ++ log2)
```

Cum compunem funcții cu efecte laterale

Problema generală

Dată fiind funcția $f :: a \rightarrow m b$ și funcția $g :: b \rightarrow m c$, vreau să obțin o funcție $g <=< f :: a \rightarrow m c$ care este „compunerea” lui g și f , propagând efectele laterale.

Exemplu

```
> logInc x = Writer  
(x + 1, "Called inc with arg " ++ show x ++ "\n")  
  
> logInc <=< logInc $ 3  
Writer {runWriter =  
(5,"Called inc with arg 3\n Called inc with arg 4\n")}
```

Observație: Funcția ($<=<$) este definită în Control.Monad

Clasa de tipuri Monad

```
class Applicative m => Monad m where
    (=>=) :: m a -> (a -> m b) -> m b
    (=>) :: m a -> m b -> m b
    return :: a -> m a
```

$ma \gg mb = ma \gg= \lambda _ \rightarrow mb$

- m este tipul **computațiilor** care produc rezultate de tip a (și au efecte laterale)
- $a \rightarrow m b$ este tipul **continuărilor** / a funcțiilor cu efecte laterale
- $\gg=$ este operația de „secvențiere” a computațiilor

Clasa de tipuri Monad

```
class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    (*)> :: m a -> m b -> m b
    return :: a -> m a
```

$ma \gg mb = ma \gg= \lambda_{} \rightarrow mb$

- m este tipul **computațiilor** care produc rezultate de tip a (și au efecte laterale)
- $a \rightarrow m b$ este tipul **continuărilor** / a funcțiilor cu efecte laterale
- $\gg=$ este operația de „secvențiere” a computațiilor
- În `Control.Monad` sunt definite
 - $f \gg= g = \lambda x \rightarrow f x \gg= g$
 - $(\ll=) = \text{flip } (\gg=)$

Clasa de tipuri Monad

```
class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    (*)> :: m a -> m b -> m b
    return :: a -> m a
```

$ma \gg mb = ma \gg= \lambda_{} \rightarrow mb$

- m este tipul **computațiilor** care produc rezultate de tip a (și au efecte laterale)
- $a \rightarrow m b$ este tipul **continuărilor** / a funcțiilor cu efecte laterale
- $\gg=$ este operația de „secvențiere” a computațiilor
- În `Control.Monad` sunt definite
 - $f \gg= g = \lambda x \rightarrow f x \gg= g$
 - $(\ll=) = \text{flip } (\gg=)$

Applicative va fi discutată mai târziu

Clasa de tipuri Monad

```
class Applicative m => Monad m where
    (=>) :: m a -> (a -> m b) -> m b
    (.) :: m a -> m b -> m b
    return :: a -> m a
```

$ma \gg mb = ma \gg= \lambda_{} \rightarrow mb$

- $m a$ este tipul **computațiilor** care produc rezultate de tip a (și au efecte laterale)
- $a \rightarrow m b$ este tipul **continuărilor** / a funcțiilor cu efecte laterale
- $\gg=$ este operația de „secvențiere” a computațiilor

În Haskell, monada este o clasă de tipuri!

Exemple: monada Maybe

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

```
> (lookup 3 [(1,2), (3,4)]) >>=
  (\x -> if (x<0) then Nothing else (Just x))
Just 4
```

```
> (lookup 3 [(1,2), (3,-4)]) >>=
  (\x -> if (x<0) then Nothing else (Just x))
Nothing
```

```
> (lookup 3 [(1,2)]) >>=
  (\x -> if (x<0) then Nothing else (Just x))
Nothing
```

Proprietățile monadelor

Asociativitate și element neutru

Operația \circ de compunere a funcțiilor îmbogățite este asociativă și are element neutru return.

Proprietățile monadelor

Asociativitate și element neutru

Operația $\leqslant\ll$ de compunere a funcțiilor îmbogățite este asociativă și are element neutru return.

- Element neutru (la dreapta): $g \leqslant\ll \text{return} = g$

$(\mathbf{return}\ x) \gg= g = g\ x$

- Element neutru (la stânga): $\text{return} \leqslant\ll g = g$

$x \gg= \mathbf{return} = x$

- Asociativitate: $h \leqslant\ll (g \leqslant\ll f) = (h \leqslant\ll g) \leqslant\ll f$

$(f \gg= g) \gg= h = f \gg= (\lambda x \rightarrow (g\ x \gg= h))$

Notația do pentru monade

Notația cu operatori	Notația do
$e >>= \lambda x \rightarrow \text{rest}$	$x \leftarrow e$ rest
$e >>= \lambda_{} \rightarrow \text{rest}$	e rest
$e >> \text{rest}$	e rest

Notația do pentru monade

Notația cu operatori	Notația do
$e >>= \lambda x \rightarrow \text{rest}$	$x \leftarrow e$ rest
$e >>= \lambda _ \rightarrow \text{rest}$	e rest
$e >> \text{rest}$	e rest

De exemplu

```
e1    >>= \x1 ->
e2    >> e3
```

devine

Notația do pentru monade

Notația cu operatori	Notația do
$e >>= \lambda x \rightarrow \text{rest}$	$x \leftarrow e$ rest
$e >>= \lambda _ \rightarrow \text{rest}$	e rest
$e >> \text{rest}$	e rest

De exemplu

```
e1    >>= \x1 ->
e2    >> e3
```

devine

do

```
x1 <- e1
e2
e3
```

Notația do pentru monade

De exemplu

```
e1    >>= \x1 ->
e2    >>= \x2 ->
e3    >>= \_  ->
e4    >>= \x4 ->
e5
```

devine

Notația do pentru monade

De exemplu

```
e1    >>= \x1 ->
e2    >>= \x2 ->
e3    >>= \_  ->
e4    >>= \x4 ->
e5
```

devine

do

```
x1 <- e1
x2 <- e2
e3
x4 <- e4
e5
```

Exemple de efecte laterale

I/O	Monada IO
Logging	Monada Writer
Stare	Monada State
Excepții	Monada Either
Partialitate	Monada Maybe
Nedeterminism	Monada [] (listă)
Memorie read-only	Monada Reader

Monada Maybe (a funcțiilor parțiale)

```
data Maybe a = Nothing | Just a
```

Monada Maybe (a funcțiilor parțiale)

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
    return = Just
    Just va  >>= k    = k va
    Nothing  >>= _    = Nothing
```

Monada Maybe (a funcțiilor parțiale)

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
    return = Just
    Just va  >>= k    = k va
    Nothing  >>= _    = Nothing

radical :: Float -> Maybe Float
radical x | x >= 0 = return (sqrt x)
          | x < 0  = Nothing
```

Monada Maybe (a funcțiilor parțiale)

```
radical :: Float -> Maybe Float
radical x | x >= 0 = return (sqrt x)
           | x < 0 = Nothing
```

```
solEq2 :: Float -> Float -> Float -> Maybe Float
solEq2 0 0 0 = return 0
-- a * x^2 + b * x + c = 0
solEq2 0 0 c = Nothing
solEq2 0 b c = return ((negate c) / b)
solEq2 a b c = do
    rDelta <- radical (b * b - 4 * a * c)
    return ((negate b + rDelta) / (2 * a))
```

Monada Either (a exceptiilor)

```
data Either err a = Left err | Right a
```

Monada Either (a exceptiilor)

```
data Either err a = Left err | Right a

instance Monad (Either err) where
    return = Right
    Right va >>= k = k va
    Left verr >>= _ = Left verr
```

Monada Either (a exceptiilor)

```
radical :: Float -> Either String Float
radical x | x >= 0 = return (sqrt x)
          | x < 0  = Left "radical: argument negativ"

solEq2 :: Float -> Float -> Float -> Either String Float
solEq2 0 0 0 = return 0
solEq2 0 0 c = Left "Nu are solutii"
solEq2 0 b c = return ((negate c) / b)
solEq2 a b c = do
    rDelta <- radical (b * b - 4 * a * c)
    return ((negate b + rDelta) / (2 * a))
```

Monada Writer

Clasa de tipuri Semigroup

O mulțime, cu o operație $\langle\rangle$ care ar trebui să fie asociativă

```
class Semigroup a where
  ( $\langle\rangle$ ) :: a -> a -> a
```

Clasa de tipuri Monoid

Un semigrup cu unitatea mempty. mappend este alias pentru $\langle\rangle$.

```
class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mappend = ( $\langle\rangle$ )
```

Monada Writer

Clasa de tipuri Functor

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Clasa de tipuri Applicative

```
class Functor m => Applicative m where
    pure :: a -> m a
    (<*>) :: m (a -> b) -> m a -> m b
```

Monada Writer

```
newtype Writer log a = Writer {runWriter :: (a, log)}
```

```
instance Functor (Writer log) where
    fmap f (Writer (a,log)) = Writer (f a,log)
```

```
instance Monoid log => Applicative (Writer log) where
    pure a = Writer (a,mempty)
    (Writer (f,log1)) <*> (Writer (a,log2)) =
        Writer (f a, log1 <> log2)
```

```
instance Monoid log => Monad (Writer log) where
    return a = Writer (a, mempty)
    ma >>= k = let (va, log1) = runWriter ma
                (vb, log2) = runWriter (k va)
            in Writer (vb, log1 <> log2)
```

Monada Writer - Exemplu logging

```
tell :: log -> Writer log ()
tell msg = Writer ((), msg)

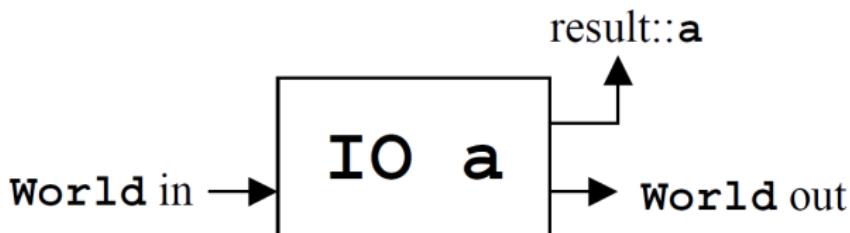
logIncrement :: Int -> Writer String Int
logIncrement x = do
    tell ("increment: " ++ show x ++ "\n")
    return (x + 1)

logIncrement2 :: Int -> Writer String Int
logIncrement2 x = do
    y <- logIncrement x
    logIncrement y

Main> runWriter (logIncrement2 13)
(15,"increment: 13\nincrement: 14\n")
```

Monads IO

```
type IO a = RealWorld -> (a, RealWorld)
```



S. Peyton-Jones, Tackling the Awkward Squad: ...

Comenzi cu valori

- `IO ()` corespunde comenziilor care nu produc rezultate

```
putChar :: Char -> IO ()  
putStr  :: String -> IO ()  
putStrLn :: String -> IO ()
```

- În general, `IO a` corespunde comenziilor care produc rezultate de tip `a`.

```
getChar :: IO Char  
getLine :: IO String
```



Pe săptămâna viitoare!

Curs 12

2021-2022

Fundamentele limbajelor de programare

Cuprins

1 Monade în Haskell (recap)

2 Interprotoare monadice

3 Introducere în λ -calcul

Monade în Haskell (recap)

Funcții îmbogățite și efecte

- Funcție simplă: $x \mapsto y$
știind x , obținem **direct** y

- Funcție îmbogățită: $x \mapsto$



știind x , putem să **extragem** y și producem un **efect**

<https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/>

<https://bartoszmilewski.com/2016/11/30/monads-and-effects/>

Funcții îmbogățite și efecte

Funcție îmbogățită: $x \mapsto$



Exemplu

- Folosind tipul Maybe a

```
data Maybe a = Nothing | Just a
f :: Int -> Maybe Int
f x = if x < 0 then Nothing else (Just x)
```

Funcții îmbogățite și efecte

Funcție îmbogățită: $x \mapsto$



Exemplu

- Folosind un tip care are ca efect un mesaj

```
newtype Writer log a = Writer {runWriter :: (a, log)}
```

```
f :: Int -> Writer String Int
```

```
f x = if x < 0 then (Writer (-x, "negativ"))
      else (Writer (x, "pozitiv"))
```

Clasa de tipuri Monad

```
class Applicative m => Monad m where
    (=>=) :: m a -> (a -> m b) -> m b
    (=>) :: m a -> m b -> m b
    return :: a -> m a
```

$ma \gg mb = ma \gg= \lambda_{} \rightarrow mb$

- $m a$ este tipul **computațiilor** care produc rezultate de tip a (și au efecte laterale)
- $a \rightarrow m b$ este tipul **continuărilor** / a funcțiilor cu efecte laterale
- $\gg=$ este operația de „secvențiere” a computațiilor

În Haskell, monada este o clasă de tipuri!

Notația do pentru monade

Notația cu operatori	Notația do
$e >>= \lambda x \rightarrow \text{rest}$	$x \leftarrow e$ rest
$e >>= \lambda _ \rightarrow \text{rest}$	e rest
$e >> \text{rest}$	e rest

De exemplu

```
e1    >>= \x1 ->
e2    >> e3
```

devine

do

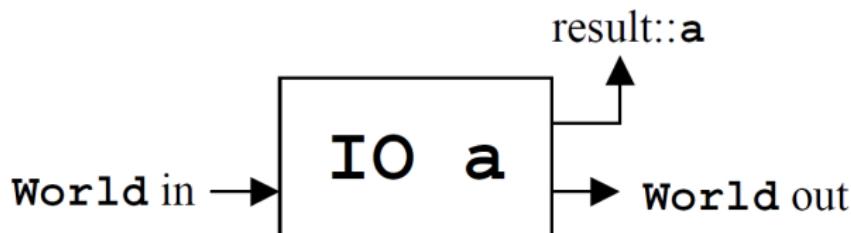
```
x1 <- e1
e2
e3
```

Exemple de efecte laterale

I/O	Monada IO
Logging	Monada Writer
Stare	Monada State
Excepții	Monada Either
Partialitate	Monada Maybe
Nedeterminism	Monada [] (listă)
Memorie read-only	Monada Reader

Monads IO

```
type IO a = RealWorld -> (a, RealWorld)
```



S. Peyton-Jones, Tackling the Awkward Squad: ...

Monada IO

- **IO ()** corespunde comenzilor care nu produc rezultate

```
putChar   :: Char -> IO ()  
putStr    :: String -> IO ()  
putStrLn  :: String -> IO ()
```

- **IO a** corespunde comenzilor care produc rezultate de tip **a**

```
getChar   :: IO Char  
getLine   :: IO String
```

Monada IO

```
sayHello :: String -> IO ()  
sayHello name = putStrLn ("Hi " ++ name ++ "!")  
  
main :: IO ()  
main = do  
    putStrLn "Please input your name: "  
    name <- getLine  
    sayHello name
```

Interprotoare monadice

Mini-Haskell - Sintaxă abstractă

```
type Name = String

data Term = Var Name
          | Con Integer
          | Term :+: Term
          | Lam Name Term
          | App Term Term
```

Mini-Haskell - Valori și medii de evaluare

```
data Value = Num Integer
           | Fun (Value -> M Value)
           | Wrong

instance Show Value where
  show (Num x) = show x
  show (Fun _) = "<function>"
  show (Wrong) = "<wrong>"
```

Observații

- Vom interpreta termenii în valori 'M Value', unde 'M' este o monadă; variind monada, se obțin comportamente diferite;
- 'Wrong' reprezintă o eroare, de exemplu adunarea unor valori care nu sunt numere sau aplicarea unui termen care nu e funcție.

Evaluare - variabile și valori

```
type Environment = [(Name, Value)]
```

Interpretarea termenilor în monada M

```
interp :: Term -> Environment -> M Value
interp (Var x) env = lookupM x env
interp (Con i) _ = return $ Num i
interp (Lam x e) env = return $
                      Fun $ \ v -> interp e ((x,v):env)
```

```
lookupM :: Name -> Environment -> M Value
```

```
lookupM x env = case lookup x env of
  Just v -> return v
  Nothing -> return Wrong
```

Evaluare - adunare

```
interp (t1 :+: t2) env = do
    v1 <- interp t1 env
    v2 <- interp t2 env
    add v1 v2
```

Interpretarea adunării în monada M

```
add :: Value -> Value -> M Value
add (Num i) (Num j) = return (Num $ i + j)
add _ _ = return Wrong
```

Evaluare - aplicarea funcțiilor

```
interp (App t1 t2) env = do
    f <- interp t1 env
    v <- interp t2 env
    apply f v
```

Interpretarea aplicării funcțiilor în monada M

```
apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply _ _ = return Wrong
-- k :: Value -> M Value
```

Testarea interpretorului

```
test :: Term -> String  
test t = showM $ interp t []
```

unde

`showM :: Show a => M a -> String`

este o funcție definită special pentru fiecare tip de efecte laterale dorit.

Testarea interpretorului

```
test :: Term -> String  
test t = showM $ interp t []
```

unde

showM :: Show a => M a -> String

este o funcție definită special pentru fiecare tip de efecte laterale dorit.

Exemplu de program

```
pgm :: Term  
pgm = App  
      (Lam "x" ((Var "x") :+: (Var "x")))  
      ((Con 10) :+: (Con 11))
```

test pgm -- apelul pentru testare

Interpreter monadic

```
data Value = Num Integer  
           | Fun (Value -> M Value)  
           | Wrong
```

```
interp :: Term -> Environment -> M Value
```

În continuare vom înlocui monada M cu:

- Identity
- Maybe
- Either String
- Writer

Interpretare în monada 'Identity'

Monada 'Identity' este "efectul identitate".

```
newtype Identity a = Identity { runIdentity :: a }
```

```
instance Monad Identity where
    return a = Identity a
    ma >>= k = k (runIdentity ma)
```

```
instance Applicative Identity where
    pure = return
    mf <*> ma = do
        f <- mf
        a <- ma
        return (f a)
```

```
instance Functor Identity where
    fmap f ma = pure f <*> ma
```

Interpretare în monada ‘Identity’

Pentru a particulariza interpretorul definim

```
type M a = Identity a
```

```
showM :: Show a => M a -> String  
showM = show . runIdentity
```

Interpretare în monada ‘Identity’

Pentru a particulariza interpretorul definim

```
type M a = Identity a
```

```
showM :: Show a => M a -> String  
showM = show . runIdentity
```

Obținem interpretorul standard, asemănător celui discutat pentru limbajul Mini-Haskell.

Interpretare folosind monada ‘Identity’

```
type M a = Identity a

showM :: Show a => M a -> String
showM = show . runIdentity

pgm = App
      (Lam "x" ((Var "x") :+: (Var "x")))
      ((Con 10) :+: (Con 11))

*Var0> test pgm
"42"
```

Interpretare în monada 'Maybe' (opțiune)

```
data Maybe a = Nothing | Just a
```

```
instance Monad Maybe where
    return = Just
    Just a >>= k = k a
    Nothing >>= _ = Nothing
```

Plus instante pentru Applicative si Functor.

Interpretare în monada ‘Maybe’ (opțiune)

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
    return = Just
    Just a >>= k = k a
    Nothing >>= _ = Nothing
```

Plus instante pentru Applicative si Functor.

Putem renunța la valoarea ‘Wrong’, folosind monada ‘Maybe’

```
type M a = Maybe a

showM :: Show a => M a -> String
showM (Just a) = show a
showM Nothing = "<wrong>"
```

Interpretare în monada ‘Maybe’

Putem acum înlocui rezultatele ‘Wrong’ cu ‘Nothing’

```
type M a = Maybe a
```

```
lookupM :: Name -> Environment -> M Value
```

```
lookupM x env = case lookup x env of
```

```
Just v -> return v
```

```
Nothing -> Nothing
```

```
add :: Value -> Value -> M Value
```

```
add (Num i) (Num j) = return (Num $ i + j)
```

```
add _ _ = Nothing
```

```
apply :: Value -> Value -> M Value
```

```
apply (Fun k) v = k v
```

```
apply _ _ = Nothing
```

Interpretare în monada ‘Either String‘

```
data Either a b = Left a | Right b

instance Monad (Either err) where
    return = Right
    Right a >>= k = k a
    err       >>= _ = err
```

Plus instante pentru Applicative si Functor.

Interpretare în monada ‘Either String’

```
data Either a b = Left a | Right b

instance Monad (Either err) where
    return = Right
    Right a >>= k = k a
    err       >>= _ = err
```

Plus instante pentru Applicative si Functor.

Putem nuanța erorile folosind monada ‘Either String’

```
type M a = Either String a

showM :: Show a => M a -> String
showM (Left s) = "Error: " ++ s
showM (Right a) = "Success: " ++ show a
```

Interpretare în monada ‘Either String’

Putem acum înlocui rezultatele ‘Wrong’ cu valori ‘Left’

```
type M a = Either String a
```

```
lookupM :: Name -> Environment -> M Value
```

```
lookupM x env = case lookup x env of
  Just v -> return v
  Nothing -> Left ("unbound variable " ++ x)
```

```
add :: Value -> Value -> M Value
```

```
add (Num i) (Num j) = return $ Num $ i + j
```

```
add v1 v2           = Left $
```

```
"Expected numbers: " ++ show v1 ++ ", " ++ show v2
```

```
apply :: Value -> Value -> M Value
```

```
apply (Fun k) v = k v
```

```
apply v _         = Left $
```

```
"Expected function: " ++ show v
```

Interpretare în monada ‘Either String‘

```
type M a = Either String a

showM :: Show a => M a -> String
showM (Left s)  = "Error: " ++ s
showM (Right a) = "Success: " ++ show a

pgm = App
      (Lam "x" ((Var "x") :+: (Var "x")))
      ((Con 10) :+: (Con 11))

*Var2> test pgm
"Success: 42"
```

Interpretare în monada ‘Either String‘

```
type M a = Either String a

showM :: Show a => M a -> String
showM (Left s)  = "Error: " ++ s
showM (Right a) = "Success: " ++ show a

pgm = App
      (Lam "x" ((Var "x") :+: (Var "x")))
      ((Con 10) :+: (Con 11))

*Var2> test pgm
"Success: 42"

pgmE = App (Var "x") ((Con 10) :+: (Con 11))

*Var2> test pgmE
"Error: unbound variable x"
```

Monada 'Writer'

Este folosită pentru a acumula (logging) informație produsă în timpul execuției.

```
newtype Writer log a = Writer { runWriter :: (a, log) }

instance Monoid log => Monad (Writer log) where
  return a = Writer (a, mempty)
  ma >>= k = let (a, log1) = runWriter ma
                (b, log2) = runWriter (k a)
  in Writer (b, log1 `mappend` log2)
```

Plus instante pentru Applicative si Functor.

Monada 'Writer'

Este folosită pentru a acumula (logging) informație produsă în timpul execuției.

```
newtype Writer log a = Writer { runWriter :: (a, log) }

instance Monoid log => Monad (Writer log) where
  return a = Writer (a, mempty)
  ma >>= k = let (a, log1) = runWriter ma
                (b, log2) = runWriter (k a)
    in Writer (b, log1 `mappend` log2)
```

Plus instante pentru Applicative si Functor.

Funcție ajutătoare

```
tell :: log -> Writer log ()
tell log = Writer ((), log) -- produce mesajul
```

Interpretare în monada ‘Writer’

Adăugarea unei instrucțiuni de afișare
data Term = ... | Out Term

```
type M a = Writer String a
```

```
showM :: Show a => M a -> String
```

```
showM ma = "Output: " ++ w ++ " Value: " ++ show a
where (a, w) = runWriter ma
```

```
interp (Out t) env = do
  v <- interp t env
  tell (show v ++ "; ")
  return v
```

- Out t se evaluează la valoarea lui t, cu efectul lateral de a adăuga valoarea la sirul de ieșire.

Interpretare în monada ‘Writer’

```
data Term = ... | Out Term

type M a = Writer String a

showM :: Show a => M a -> String
showM ma = "Output: " ++ w ++ " Value: " ++ show a
where (a, w) = runWriter ma

pgmW = App
    (Lam "x" ((Var "x") :+: (Var "x")))
    ((Out (Con 10)) :+: (Out (Con 11)))

> test pgm
"Output: 10; 11; Value: 42"
```

Introducere în λ -calcul

λ -calcul

- În 1929-1932 Church a propus λ -calculul ca sistem formal pentru logica matematică. În 1935 a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată în λ -calcul.
- În 1935, independent de Church, Turing a dezvoltat mecanismul de calcul numit astăzi Mașina Turing. În 1936 și el a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată de o mașină Turing. De asemenea, a arătat echivalența celor două modele de calcul. Această echivalență a constituit o indicație puternică asupra "universalității" celor două modele, conducând la ceea ce numim astăzi "Teza Church-Turing".

Referințe

- Benjamin C. Pierce, Types and Programming Languages, The MIT Press 2002
- J.R. Hindley, J.P. Seldin, Lambda-Calculus and Combinators, an Introduction, Cambridge University Press, 2008
- R. Nederpelt, H. Geuvers, Type Theory and Formal Proof, an Introduction, Cambridge University Press 2014

λ -calcul: sintaxă

Lambda Calcul - sintaxă

$t = \begin{array}{ll} x & \text{(variabilă)} \\ | \lambda x. t & \text{(abstractizare)} \\ | t\ t & \text{(aplicare)} \end{array}$

λ -calcul: sintaxă

Lambda Calcul - sintaxă

$$\begin{aligned} t = & \quad x \quad \text{(variabilă)} \\ & | \lambda x. t \quad \text{(abstractizare)} \\ & | t t \quad \text{(aplicare)} \end{aligned}$$

λ -termeni

Fie $Var = \{x, y, z, \dots\}$ o mulțime infinită de variabile.

Mulțimea λ -termenilor ΛT este definită inductiv astfel:

[Variabilă] $Var \subseteq \Lambda T$

[Aplicare] dacă $t_1, t_2 \in \Lambda T$ atunci $(t_1 t_2) \in \Lambda T$

[Abstractizare] dacă $x \in Var$ și $t \in \Lambda T$ atunci $(\lambda x. t) \in \Lambda T$

Lambda termeni

λ -termeni: exemple

- x, y, z

Lambda termeni

λ -termeni: exemple

- x, y, z
- $(xy), (yx), (x(yx))$

Lambda termeni

λ -termeni: exemple

- x, y, z
- $(xy), (yx), (x(yx))$
- $(\lambda x.x), (\lambda x.(xy)), (\lambda z.(xy)), (\lambda x.(\lambda z.(xy)))$

Lambda termeni

λ -termeni: exemple

- x, y, z
- $(xy), (yx), (x(yx))$
- $(\lambda x.x), (\lambda x.(xy)), (\lambda z.(xy)), (\lambda x.(\lambda z.(xy)))$
- $((\lambda x.x)y), ((\lambda x.(xz))y), ((\lambda x.x)(\lambda y.y))$

Lambda termeni

λ -termeni: exemple

- x, y, z
- $(xy), (yx), (x(yx))$
- $(\lambda x.x), (\lambda x.(xy)), (\lambda z.(xy)), (\lambda x.(\lambda z.(xy)))$
- $((\lambda x.x)y), ((\lambda x.(xz))y), ((\lambda x.x)(\lambda y.y))$

Convenții:

- se elimină parantezele exterioare
- aplicarea este asociativă la stânga: $t_1 t_2 t_3$ este $(t_1 t_2) t_3$
- corpul abstractizării este extins la dreapta:
 $\lambda x.t_1 t_2$ este $\lambda x.(t_1 t_2)$ (nu $(\lambda x.t_1)t_2$)
- scriem $\lambda xyz.t$ în loc de $\lambda x.\lambda y.\lambda z.t$

Lambda termeni. Funcții anonte

λ -termeni: exemple

- x, y, z
- $(xy), (yx), (x(yx))$
- $(\lambda x.x), (\lambda x.(xy)), (\lambda z.(xy)), (\lambda x.(\lambda z.(xy)))$
- $((\lambda x.x)y), ((\lambda x.(xz))y), ((\lambda x.x)(\lambda y.y))$

Funcții anonte în Haskell

În Haskell, \ e folosit în locul simbolului λ și \rightarrow în locul punctului.

$\lambda x.x * x$ este $\backslash x \rightarrow x * x$

$\lambda x.x > 0$ este $\backslash x \rightarrow x > 0$

Variabile libere și legate

Apariții libere și legate

Pentru un termen $\lambda x.t$ spunem că:

- aparițiile variabilei x în t sunt **legate** (*bound*)
- λx este **legătura** (*binder*), iar t este domeniul (*scope*) legării
- o apariție a unei variabile este **liberă** (*free*) dacă apare într-o poziție în care nu e legată.

Variabile libere și legate

Apariții libere și legate

Pentru un termen $\lambda x.t$ spunem că:

- aparițiile variabilei x în t sunt **legate** (*bound*)
- λx este **legătura** (*binder*), iar t este domeniul (*scope*) legării
- o apariție a unei variabile este **liberă** (*free*) dacă apare într-o poziție în care nu e legată.

Un termen fără variabile libere se numește **închis** (*closed*).

Variabile libere și legate

Apariții libere și legate

Pentru un termen $\lambda x.t$ spunem că:

- aparițiile variabilei x în t sunt **legate** (*bound*)
- λx este **legătura** (*binder*), iar t este domeniul (*scope*) legării
- o apariție a unei variabile este **liberă** (*free*) dacă apare într-o poziție în care nu e legată.

Un termen fără variabile libere se numește **închis** (*closed*).

Exemplu:

- $\lambda x.x$ este un termen închis
- $\lambda x.xy$ nu este termen închis, x este legată, y este liberă
- în termenul $x(\lambda x.xy)$ prima apariție a lui x este liberă, a doua este legată.

Variabile libere

Mulțimea variabilelor libere $FV(t)$

Pentru un λ -termen t mulțimea variabilelor libere este definită astfel:

[Variabilă] $FV(x) = \{x\}$

[Aplicare] $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$

[Abstractizare] $FV(\lambda x. t) = FV(t) \setminus \{x\}$

Variabile libere

Mulțimea variabilelor libere $FV(t)$

Pentru un λ -termen t mulțimea variabilelor libere este definită astfel:

[Variabilă] $FV(x) = \{x\}$

[Aplicare] $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$

[Abstractizare] $FV(\lambda x. t) = FV(t) \setminus \{x\}$

Exemplu:

$$\begin{aligned} FV(\lambda x. xy) &= FV(xy) \setminus \{x\} \\ &= (FV(x) \cup FV(y)) \setminus \{x\} \\ &= (\{x\} \cup \{y\}) \setminus \{x\} \\ &= \{y\} \end{aligned}$$

Variabile libere

Mulțimea variabilelor libere $FV(t)$

Pentru un λ -termen t mulțimea variabilelor libere este definită astfel:

[Variabilă] $FV(x) = \{x\}$

[Aplicare] $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$

[Abstractizare] $FV(\lambda x. t) = FV(t) \setminus \{x\}$

Exemplu:

$$\begin{aligned} FV(\lambda x. xy) &= FV(xy) \setminus \{x\} \\ &= (FV(x) \cup FV(y)) \setminus \{x\} \\ &= (\{x\} \cup \{y\}) \setminus \{x\} \\ &= \{y\} \end{aligned}$$

$$FV(x\lambda x. xy) =$$

Variabile libere

Mulțimea variabilelor libere $FV(t)$

Pentru un λ -termen t mulțimea variabilelor libere este definită astfel:

[Variabilă] $FV(x) = \{x\}$

[Aplicare] $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$

[Abstractizare] $FV(\lambda x. t) = FV(t) \setminus \{x\}$

Exemplu:

$$\begin{aligned} FV(\lambda x. xy) &= FV(xy) \setminus \{x\} \\ &= (FV(x) \cup FV(y)) \setminus \{x\} \\ &= (\{x\} \cup \{y\}) \setminus \{x\} \\ &= \{y\} \end{aligned}$$

$$FV(x\lambda x. xy) = \{x, y\}$$

Substituții

Fie t un λ -termen și $x \in \text{Var}$.

Definiție intuitivă

Pentru un λ -termen u vom nota prin $[u/x]t$ rezultatul înlocuirii tuturor aparițiilor libere ale lui x cu u în t .

Substituții

Fie t un λ -termen și $x \in \text{Var}$.

Definiție intuitivă

Pentru un λ -termen u vom nota prin $[u/x]t$ rezultatul înlocuirii tuturor aparițiilor libere ale lui x cu u în t .

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$
- $[(\lambda z.zw)/x](\lambda y.x) = \lambda y.\lambda z.zw$

Substituții

Definirea substituției

Rezultatul substituirii lui x cu u în t este definit astfel:

[Variabilă] $[u/x]x = u$

[Variabilă] $[u/x]y = y$ dacă $x \neq y$

[Aplicare] $[u/x](t_1 t_2) = [u/x]t_1 [u/x]t_2$

[Abstractizare] $[u/x]\lambda y.t = \lambda y.[u/x]t$ unde
 $y \neq x$ și $y \notin FV(u)$

Substituții

Exemple: Dacă x, y, z sunt variabile distincte atunci

□ $[y/x]\lambda z.x = \lambda z.y$

Substituții

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$
- Cine este $[y/x]\lambda y.x$?

Substituții

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$
- Cine este $[y/x]\lambda y.x$?

Dacă folosim definiția intuitivă obținem

$[y/x]\lambda y.x = \lambda y.y$ ceea ce este **greșit!**

Substituții

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$
- Cine este $[y/x]\lambda y.x$?

Dacă folosim definiția intuitivă obținem

$[y/x]\lambda y.x = \lambda y.y$ ceea ce este **greșit!**

Cum procedăm pentru a repara greșeala? Observăm că $\lambda y.x$ desemnează o funcție constantă, aceeași funcție putând fi reprezentată prin $\lambda z.x$. Aplicarea **corectă** a substituției este:

$$[y/x]\lambda y.x = [y/x]\lambda z.x = \lambda z.y$$

Avem libertatea de a redenumi variabilele legate!

α -conversie (α -echivalență)

Definim următoarea relație peste termeni:

α -conversia $=_{\alpha}$

α -conversie (α -echivalență)

Definim următoarea relație peste termeni:

α -conversia $=_{\alpha}$

[Reflexivitate] $t =_{\alpha} t$

α -conversie (α -echivalență)

Definim următoarea relație peste termeni:

α -conversia $=_{\alpha}$

[Reflexivitate] $t =_{\alpha} t$

[Simetrie] $t_1 =_{\alpha} t_2$ implică $t_2 =_{\alpha} t_1$

α -conversie (α -echivalență)

Definim următoarea relație peste termeni:

α -conversia $=_{\alpha}$

[Reflexivitate] $t =_{\alpha} t$

[Simetrie] $t_1 =_{\alpha} t_2$ implică $t_2 =_{\alpha} t_1$

[Tranzitivitate] $t_1 =_{\alpha} t_2$ și $t_2 =_{\alpha} t_3$ implică $t_1 =_{\alpha} t_3$

α -conversie (α -echivalență)

Definim următoarea relație peste termeni:

α -conversia $=_{\alpha}$

[Reflexivitate] $t =_{\alpha} t$

[Simetrie] $t_1 =_{\alpha} t_2$ implică $t_2 =_{\alpha} t_1$

[Tranzitivitate] $t_1 =_{\alpha} t_2$ și $t_2 =_{\alpha} t_3$ implică $t_1 =_{\alpha} t_3$

[Redenumire] $\lambda x.t =_{\alpha} \lambda y.[y/x]t$ dacă $y \notin FV(t)$

α -conversie (α -echivalență)

Definim următoarea relație peste termeni:

α -conversia $=_{\alpha}$

[Reflexivitate] $t =_{\alpha} t$

[Simetrie] $t_1 =_{\alpha} t_2$ implică $t_2 =_{\alpha} t_1$

[Tranzitivitate] $t_1 =_{\alpha} t_2$ și $t_2 =_{\alpha} t_3$ implică $t_1 =_{\alpha} t_3$

[Redenumire] $\lambda x.t =_{\alpha} \lambda y.[y/x]t$ dacă $y \notin FV(t)$

[Compatibilitate] $t_1 =_{\alpha} t_2$ implică

$t t_1 =_{\alpha} t t_2, t_1 t =_{\alpha} t_2 t$ și $\lambda x.t_1 =_{\alpha} \lambda x.t_2$

α -conversie (α -echivalență)

Definim următoarea relație peste termeni:

α -conversia $=_{\alpha}$

[Reflexivitate] $t =_{\alpha} t$

[Simetrie] $t_1 =_{\alpha} t_2$ implică $t_2 =_{\alpha} t_1$

[Tranzitivitate] $t_1 =_{\alpha} t_2$ și $t_2 =_{\alpha} t_3$ implică $t_1 =_{\alpha} t_3$

[Redenumire] $\lambda x.t =_{\alpha} \lambda y.[y/x]t$ dacă $y \notin FV(t)$

[Compatibilitate] $t_1 =_{\alpha} t_2$ implică

$t t_1 =_{\alpha} t t_2$, $t_1 t =_{\alpha} t_2 t$ și $\lambda x.t_1 =_{\alpha} \lambda x.t_2$

Exemplu:

$$[xy/x](\lambda y.yx) =_{\alpha} [xy/x](\lambda z.zx) =_{\alpha} \lambda z.z(xy)$$

α -conversie (α -echivalență)

Definim următoarea relație peste termeni:

α -conversia $=_{\alpha}$

[Reflexivitate] $t =_{\alpha} t$

[Simetrie] $t_1 =_{\alpha} t_2$ implică $t_2 =_{\alpha} t_1$

[Tranzitivitate] $t_1 =_{\alpha} t_2$ și $t_2 =_{\alpha} t_3$ implică $t_1 =_{\alpha} t_3$

[Redenumire] $\lambda x.t =_{\alpha} \lambda y.[y/x]t$ dacă $y \notin FV(t)$

[Compatibilitate] $t_1 =_{\alpha} t_2$ implică

$t_1 t =_{\alpha} t_2 t$, $t_1 t =_{\alpha} t_2 t$ și $\lambda x.t_1 =_{\alpha} \lambda x.t_2$

Exemplu:

$$[xy/x](\lambda y.yx) =_{\alpha} [xy/x](\lambda z.zx) =_{\alpha} \lambda z.z(xy)$$

Vom lucra **modulo α -conversie**, doi termeni α -echivalenți vor fi considerați "egali".

α -conversie

Exemplu:

$$\square [xy/x]\lambda x.yx =_{\alpha} [xy/x]\lambda z.yz = \lambda z.[xy/x](yz) = \lambda z.yz$$

Observăm că $\lambda z.yz =_{\alpha} \lambda x.yx$

α -conversie

Exemplu:

□ $[xy/x]\lambda x.yx =_{\alpha} [xy/x]\lambda z.yz = \lambda z.[xy/x](yz) = \lambda z.yz$

Observăm că $\lambda z.yz =_{\alpha} \lambda x.yx$

□ $[y/z]\lambda xy.zzx = \lambda x.[y/z]\lambda y.zzx =_{\alpha} \lambda x.[y/z]\lambda v.zzx =$
 $\lambda x.\lambda v.[y/z](zvx) = \lambda xv.yyx$

β -reducție

β -reducția este o relație pe mulțimea α -termenilor.

β -reducția \rightarrow_{β} , $\overset{*}{\rightarrow}_{\beta}$

□ un singur pas $\rightarrow_{\beta} \subseteq \Lambda T \times \Lambda T$

[Aplicarea] $(\lambda x.t)u \rightarrow_{\beta} [u/x]t$

[Compatibilitatea] $t_1 \rightarrow_{\beta} t_2$ implica

$$t_1 t \rightarrow_{\beta} t_2 t, t_1 t \rightarrow_{\beta} t_2 t \text{ și } \lambda x. t_1 \rightarrow_{\beta} \lambda x. t_2$$

β -reducție

β -reducția este o relație pe mulțimea α -termenilor.

β -reducția \rightarrow_{β} , $\xrightarrow{\beta}^*$

- un singur pas $\rightarrow_{\beta} \subseteq \Lambda T \times \Lambda T$

[Aplicarea] $(\lambda x.t)u \rightarrow_{\beta} [u/x]t$

[Compatibilitatea] $t_1 \rightarrow_{\beta} t_2$ implica

$$t \; t_1 \rightarrow_{\beta} \; t \; t_2, \; t_1 t \rightarrow_{\beta} t_2 t \text{ și } \lambda x. t_1 \rightarrow_{\beta} \lambda x. t_2$$

- zero sau mai mulți pași $\xrightarrow{\beta}^* \subseteq \Lambda T \times \Lambda T$

$t_1 \xrightarrow{\beta}^* t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât

$$t_1 =_{\alpha} u_0 \rightarrow_{\beta} u_1 \rightarrow_{\beta} \cdots \rightarrow_{\beta} u_n =_{\alpha} t_2$$

β -reducție

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

□ $(\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda y.yv)z \rightarrow_{\beta} zv$

β -reducție

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

- $(\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda y.yv)z \rightarrow_{\beta} zv$
- $(\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda x.zx)v \rightarrow_{\beta} zv$

β -reducție

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

- $(\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda y.yv)z \rightarrow_{\beta} zv$
- $(\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda x.zx)v \rightarrow_{\beta} zv$

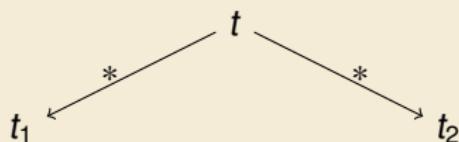
Observăm că un termen poate fi β -redus în mai multe moduri.

Proprietatea de **confluentă** ne asigură că vom ajunge întotdeauna la același rezultat.

Confluența β -reducției

Teorema Church-Rosser

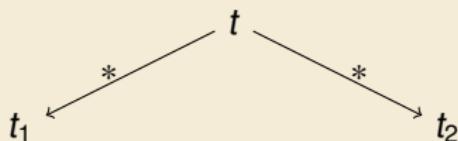
Dacă $t \xrightarrow{\beta}^* t_1$ și $t \xrightarrow{\beta}^* t_2$



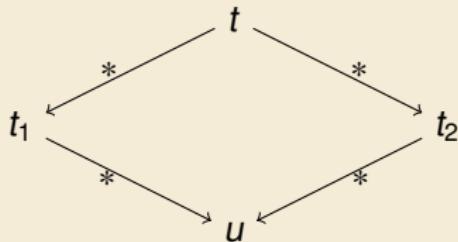
Confluența β -reducției

Teorema Church-Rosser

Dacă $t \xrightarrow{*_{\beta}} t_1$ și $t \xrightarrow{*_{\beta}} t_2$



atunci există u astfel încât $t_1 \xrightarrow{*_{\beta}} u$ și $t_2 \xrightarrow{*_{\beta}} u$.



β -forma normală

Intuitiv, o formă normală este un termen care nu mai poate fi redus (sau punctul final al unui calcul).

Formă normală

- un λ -termen căruia nu îl se mai poate aplica reducerea într-un pas \rightarrow_β se numește β -formă normală
- dacă $t \xrightarrow{*_\beta} u_1$, $t \xrightarrow{*_\beta} u_2$ și u_1 , u_2 sunt β -forme normale atunci, datorită confluenței, $u_1 =_\alpha u_2$
- un λ -termen poate avea cel mult o β -formă normală (modulo α -echivalență)

β -forma normală

Intuitiv, o formă normală este un termen care nu mai poate fi redus (sau punctul final al unui calcul).

Formă normală

- un λ -termen căruia nu i se mai poate aplica reducerea într-un pas \rightarrow_β se numește β -formă normală
- dacă $t \xrightarrow{*_\beta} u_1$, $t \xrightarrow{*_\beta} u_2$ și u_1 , u_2 sunt β -forme normale atunci, datorită confluenței, $u_1 =_\alpha u_2$
- un λ -termen poate avea cel mult o β -formă normală (modulo α -echivalență)

Exemplu:

- zv este β -formă normală pentru $(\lambda x.(\lambda y.yx)z)v$
 $(\lambda x.(\lambda y.yx)z)v \rightarrow_\beta (\lambda y.yv)z \rightarrow_\beta zv$
- există termeni care **nu** pot fi reduși la o β -formă normală, de exemplu $(\lambda x.xx)(\lambda x.xx)$

β -conversia

Intuitiv, β -conversia extinde β -reducția în ambele direcții.

□ $(\lambda y.yv)z \rightarrow_{\beta} zv \leftarrow_{\beta} (\lambda x.zx)v$

β -conversia

Intuitiv, β -conversia extinde β -reducția în ambele direcții.

- $(\lambda y.yv)z \rightarrow_{\beta} zv \leftarrow_{\beta} (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_{\beta} (\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda x.zx)v$

β -conversia

Intuitiv, β -conversia extinde β -reducția în ambele direcții.

- $(\lambda y.yv)z \rightarrow_{\beta} zv \leftarrow_{\beta} (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_{\beta} (\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda x.zx)v$

β -conversia $=_{\beta}$

- $=_{\beta} \subseteq \Lambda T \times \Lambda T$
 $t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
 $t_1 =_{\alpha} u_0, u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$

β -conversia

Intuitiv, β -conversia extinde β -reducția în ambele direcții.

- $(\lambda y.yv)z \rightarrow_{\beta} zv \leftarrow_{\beta} (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_{\beta} (\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda x.zx)v$

β -conversia $=_{\beta}$

- $=_{\beta} \subseteq \Lambda T \times \Lambda T$
 $t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
 $t_1 =_{\alpha} u_0, u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$

Exemplu: $(\lambda y.yv)z =_{\beta} (\lambda x.zx)v$

β -conversia

β -conversia $=_{\beta}$

- $=_{\beta} \subseteq \Lambda T \times \Lambda T$
- $t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
 $t_1 =_{\alpha} u_0, u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$

β -conversia

β -conversia $=_{\beta}$

- $=_{\beta} \subseteq \Lambda T \times \Lambda T$
 $t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
 $t_1 =_{\alpha} u_0, u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$

Observații

- $=_{\beta}$ este o relație de echivalență

β -conversia

β -conversia $=_{\beta}$

- $=_{\beta} \subseteq \Lambda T \times \Lambda T$
- $t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
 $t_1 =_{\alpha} u_0, u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$

Observații

- $=_{\beta}$ este o relație de echivalență
- pentru $t_1, t_2 \lambda$ -termeni și $u_1, u_2 \beta$ -forme normale
dacă $t_1 \xrightarrow{\beta}^* u_1, t_2 \xrightarrow{\beta}^* u_2$ și $u_1 =_{\alpha} u_2$ atunci $t_1 =_{\beta} t_2$

β -conversia

β -conversia $=_{\beta}$

- $=_{\beta} \subseteq \Lambda T \times \Lambda T$
- $t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
 $t_1 =_{\alpha} u_0, u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$

Observații

- $=_{\beta}$ este o relație de echivalență
- pentru $t_1, t_2 \lambda$ -termeni și $u_1, u_2 \beta$ -forme normale
dacă $t_1 \xrightarrow{\beta}^* u_1, t_2 \xrightarrow{\beta}^* u_2$ și $u_1 =_{\alpha} u_2$ atunci $t_1 =_{\beta} t_2$

β -conversia reprezintă "egalitatea prin calcul", iar β -reducția (modulo α -conversie) oferă o procedură de decizie pentru aceasta.



Pe săptămâna viitoare!

Curs 13

2021-2022

Fundamentele limbajelor de programare

Cuprins

λ -calcul

λ -calcul: sintaxă

Lambda Calcul - sintaxă

$$\begin{aligned} t = & \quad x \quad \text{(variabilă)} \\ & | \lambda x. t \quad \text{(abstractizare)} \\ & | t t \quad \text{(aplicare)} \end{aligned}$$

λ -termeni

Fie $Var = \{x, y, z, \dots\}$ o mulțime infinită de variabile.

Mulțimea λ -termenilor ΛT este definită inductiv astfel:

[Variabilă] $Var \subseteq \Lambda T$

[Aplicare] dacă $t_1, t_2 \in \Lambda T$ atunci $(t_1 t_2) \in \Lambda T$

[Abstractizare] dacă $x \in Var$ și $t \in \Lambda T$ atunci $(\lambda x. t) \in \Lambda T$

Lambda termeni

Convenții:

- se elimină parantezele exterioare
- aplicarea este asociativă la stânga: $t_1 t_2 t_3$ este $(t_1 t_2) t_3$
- corpul abstractizării este extins la dreapta:
 $\lambda x. t_1 t_2$ este $\lambda x. (t_1 t_2)$ (nu $(\lambda x. t_1) t_2$)
- scriem $\lambda xyz. t$ în loc de $\lambda x. \lambda y. \lambda z. t$

Mulțimea variabilelor libere $FV(t)$

Pentru un λ -termen t mulțimea variabilelor libere este definită astfel:

$$[\text{Variabilă}] FV(x) = \{x\}$$

$$[\text{Aplicare}] FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$$

$$[\text{Abstractizare}] FV(\lambda x. t) = FV(t) \setminus \{x\}$$

Substituții

Fie t un λ -termen și $x \in \text{Var}$.

Definiție intuitivă

Pentru un λ -termen u vom nota prin $[u/x]t$ rezultatul înlocuirii tuturor aparițiilor libere ale lui x cu u în t .

Definirea substituției

Rezultatul substituirii lui x cu u în t este definit astfel:

[Variabilă] $[u/x]x = u$

[Variabilă] $[u/x]y = y$ dacă $x \neq y$

[Aplicare] $[u/x](t_1 t_2) = [u/x]t_1 [u/x]t_2$

[Abstractizare] $[u/x]\lambda y.t = \lambda y.[u/x]t$ unde $y \neq x$ și $y \notin FV(u)$

α -conversie (α -echivalență)

Definim următoarea relație peste termeni:

α -conversia $=_{\alpha}$

[Reflexivitate] $t =_{\alpha} t$

[Simetrie] $t_1 =_{\alpha} t_2$ implică $t_2 =_{\alpha} t_1$

[Tranzitivitate] $t_1 =_{\alpha} t_2$ și $t_2 =_{\alpha} t_3$ implică $t_1 =_{\alpha} t_3$

[Redenumire] $\lambda x. t =_{\alpha} \lambda y. [y/x]t$ dacă $y \notin FV(t)$

[Compatibilitate] $t_1 =_{\alpha} t_2$ implică

$$t t_1 =_{\alpha} t t_2, t_1 t =_{\alpha} t_2 t \text{ și } \lambda x. t_1 =_{\alpha} \lambda x. t_2$$

Vom lucra **modulo α -conversie**, doi termeni α -echivalenți vor fi considerați "egali".

Dacă $t_1 =_{\alpha} t_2$, atunci vom folosi notația $[u/x]t_1 =_{\alpha} [u/x]t_2$ pentru a semnala faptul că în t_1 am redenumit variabilele.

α -conversie

Exemplu:

- $[xy/x](\lambda y.yx) =_{\alpha} [xy/x](\lambda z.zx) = \lambda z.z(xy)$
- $[xy/x]\lambda x.yx =_{\alpha} [xy/x]\lambda z.yz = \lambda z.[xy/x](yz) = \lambda z.yz$

Observăm că $\lambda z.yz =_{\alpha} \lambda x.yx$

α -conversie

Exemplu:

- $[xy/x](\lambda y.yx) =_{\alpha} [xy/x](\lambda z.zx) = \lambda z.z(xy)$
- $[xy/x]\lambda x.yx =_{\alpha} [xy/x]\lambda z.yz = \lambda z.[xy/x](yz) = \lambda z.yz$

Observăm că $\lambda z.yz =_{\alpha} \lambda x.yx$

- $[y/z]\lambda xy.zzx = \lambda x.[y/z]\lambda y.zzx =_{\alpha} \lambda x.[y/z]\lambda v.zzx = \lambda x.\lambda v.[y/z](zzx) = \lambda xv.yyx$

β -reducție

β -reducția este o relație pe mulțimea λ -termenilor, lucrând modula α -echivalență.

β -reducția \rightarrow_{β} , $\overset{*}{\rightarrow}_{\beta}$

□ un singur pas $\rightarrow_{\beta} \subseteq \Lambda T \times \Lambda T$

[Aplicarea] $(\lambda x.t)u \rightarrow_{\beta} [u/x]t$

[Compatibilitatea] $t_1 \rightarrow_{\beta} t_2$ implică

$$t_1 t \rightarrow_{\beta} t_2 t, t_1 t \rightarrow_{\beta} t_2 t \text{ și } \lambda x. t_1 \rightarrow_{\beta} \lambda x. t_2$$

β -reducție

β -reducția este o relație pe mulțimea λ -termenilor, lucrând modula α -echivalență.

β -reducția \rightarrow_{β} , $\overset{*}{\rightarrow}_{\beta}$

- un singur pas $\rightarrow_{\beta} \subseteq \Lambda T \times \Lambda T$

[Aplicarea] $(\lambda x.t)u \rightarrow_{\beta} [u/x]t$

[Compatibilitatea] $t_1 \rightarrow_{\beta} t_2$ implică

$$t_1 t_1 \rightarrow_{\beta} t_1 t_2, t_1 t \rightarrow_{\beta} t_2 t \text{ și } \lambda x. t_1 \rightarrow_{\beta} \lambda x. t_2$$

- zero sau mai mulți pași $\overset{*}{\rightarrow}_{\beta} \subseteq \Lambda T \times \Lambda T$

$t_1 \overset{*}{\rightarrow}_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât

$$t_1 =_{\alpha} u_0 \rightarrow_{\beta} u_1 \rightarrow_{\beta} \cdots \rightarrow_{\beta} u_n =_{\alpha} t_2$$

β -reducție

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

□ $(\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda y.yv)z \rightarrow_{\beta} zv$

β -reducție

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

- $(\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda y.yv)z \rightarrow_{\beta} zv$
- $(\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda x.zx)v \rightarrow_{\beta} zv$

β -reducție

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

- $(\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda y.yv)z \rightarrow_{\beta} zv$
- $(\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda x.zx)v \rightarrow_{\beta} zv$

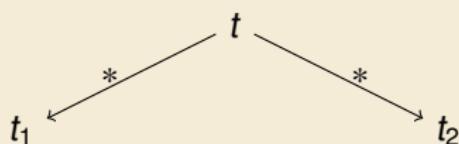
Observăm că un termen poate fi β -redus în mai multe moduri.

Proprietatea de **confluentă** ne asigură că vom ajunge întotdeauna la același rezultat.

Confluența β -reducției

Teorema Church-Rosser

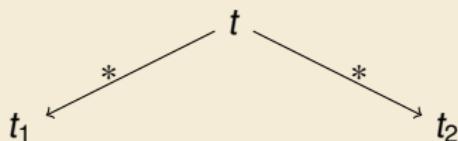
Dacă $t \xrightarrow{\beta}^* t_1$ și $t \xrightarrow{\beta}^* t_2$



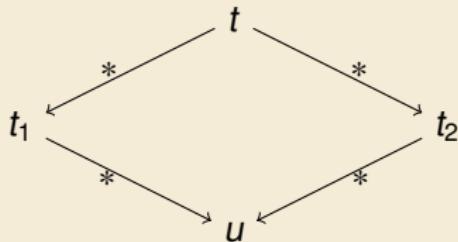
Confluența β -reducției

Teorema Church-Rosser

Dacă $t \xrightarrow{\beta}^* t_1$ și $t \xrightarrow{\beta}^* t_2$



atunci există u astfel încât $t_1 \xrightarrow{\beta}^* u$ și $t_2 \xrightarrow{\beta}^* u$.



β -forma normală

Intuitiv, o formă normală este un termen care nu mai poate fi redus (sau punctul final al unui calcul).

Formă normală

- un λ -termen căruia nu îl se mai poate aplica reducerea într-un pas \rightarrow_β se numește β -formă normală
- dacă $t \xrightarrow{*_\beta} u_1$, $t \xrightarrow{*_\beta} u_2$ și u_1, u_2 sunt β -forme normale atunci, datorită confluenței, $u_1 =_\alpha u_2$
- un λ -termen poate avea cel mult o β -formă normală (modulo α -echivalență)

β -forma normală

Intuitiv, o formă normală este un termen care nu mai poate fi redus (sau punctul final al unui calcul).

Formă normală

- un λ -termen căruia nu i se mai poate aplica reducerea într-un pas \rightarrow_β se numește β -formă normală
- dacă $t \xrightarrow{*_\beta} u_1$, $t \xrightarrow{*_\beta} u_2$ și u_1, u_2 sunt β -forme normale atunci, datorită confluenței, $u_1 =_\alpha u_2$
- un λ -termen poate avea cel mult o β -formă normală (modulo α -echivalență)

Exemplu:

- zv este β -formă normală pentru $(\lambda x.(\lambda y.yx)z)v$
 $(\lambda x.(\lambda y.yx)z)v \rightarrow_\beta (\lambda y.yv)z \rightarrow_\beta zv$
- există termeni care **nu** pot fi reduși la o β -formă normală, de exemplu $(\lambda x.xx)(\lambda x.xx)$

β -conversia

Intuitiv, β -conversia extinde β -reducția în ambele direcții.

□ $(\lambda y.yv)z \rightarrow_{\beta} zv \leftarrow_{\beta} (\lambda x.zx)v$

β -conversia

Intuitiv, β -conversia extinde β -reducția în ambele direcții.

- $(\lambda y.yv)z \rightarrow_{\beta} zv \leftarrow_{\beta} (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_{\beta} (\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda x.zx)v$

β -conversia

Intuitiv, β -conversia extinde β -reducția în ambele direcții.

- $(\lambda y.yv)z \rightarrow_{\beta} zv \leftarrow_{\beta} (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_{\beta} (\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda x.zx)v$

β -conversia $=_{\beta}$

- $=_{\beta} \subseteq \Lambda T \times \Lambda T$
 $t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
 $t_1 =_{\alpha} u_0, u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$

β -conversia

Intuitiv, β -conversia extinde β -reducția în ambele direcții.

- $(\lambda y.yv)z \rightarrow_{\beta} zv \leftarrow_{\beta} (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_{\beta} (\lambda x.(\lambda y.yx)z)v \rightarrow_{\beta} (\lambda x.zx)v$

β -conversia $=_{\beta}$

- $=_{\beta} \subseteq \Lambda T \times \Lambda T$
 $t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
 $t_1 =_{\alpha} u_0, u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$

Exemplu: $(\lambda y.yv)z =_{\beta} (\lambda x.zx)v$

β -conversia

β -conversia $=_{\beta}$

- $=_{\beta} \subseteq \Lambda T \times \Lambda T$
- $t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
 $t_1 =_{\alpha} u_0, u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$

β -conversia

β -conversia $=_{\beta}$

- $=_{\beta} \subseteq \Lambda T \times \Lambda T$
 $t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
 $t_1 =_{\alpha} u_0, u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$

Observații

- $=_{\beta}$ este o relație de echivalență

β -conversia

β -conversia $=_{\beta}$

- $=_{\beta} \subseteq \Lambda T \times \Lambda T$
- $t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
- $t_1 =_{\alpha} u_0, u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$

Observații

- $=_{\beta}$ este o relație de echivalență
- pentru t_1, t_2 λ -termeni și u_1, u_2 β -forme normale
dacă $t_1 \xrightarrow{\beta}^* u_1, t_2 \xrightarrow{\beta}^* u_2$ și $u_1 =_{\alpha} u_2$ atunci $t_1 =_{\beta} t_2$

β -conversia

β -conversia $=_{\beta}$

- $=_{\beta} \subseteq \Lambda T \times \Lambda T$
- $t_1 =_{\beta} t_2$ dacă există $n \geq 0$ și u_0, \dots, u_n astfel încât
 $t_1 =_{\alpha} u_0, u_n =_{\alpha} t_2$ și, pentru orice i , $u_i \rightarrow_{\beta} u_{i+1}$ sau $u_{i+1} \rightarrow_{\beta} u_i$

Observații

- $=_{\beta}$ este o relație de echivalență
- pentru $t_1, t_2 \lambda$ -termeni și $u_1, u_2 \beta$ -forme normale
dacă $t_1 \xrightarrow{\beta}^* u_1, t_2 \xrightarrow{\beta}^* u_2$ și $u_1 =_{\alpha} u_2$ atunci $t_1 =_{\beta} t_2$

β -conversia reprezintă "egalitatea prin calcul", iar β -reducția (modulo α -conversie) oferă o procedură de decizie pentru aceasta.

Expresivitatea λ -calculului

Expresivitatea λ -calculului

Vom arăta cum putem exprima în lambda calcul

- Booleeni
- Numere naturale

Expresivitatea λ -calculului

Vom arăta cum putem exprima în lambda calcul

- Booleeni
- Numere naturale

Intuiție: tipurile de date sunt codificate de capabilități.

- Booleeni - capabilitatea de a alege între două alternative
- Numere naturale - capabilitatea de a itera de un număr dat de ori

Booleeni

Intuitie: Capabilitatea de a alege între două alternative.

Codificare: Un Boolean este o funcție cu 2 argumente reprezentând ramurile unei alegeri.

Booleeni

Intuitie: Capabilitatea de a alege între două alternative.

Codificare: Un Boolean este o funcție cu 2 argumente reprezentând ramurile unei alegeri.

Incepem prin a defini doi λ -termeni pentru a coda "true" și "false":

- T** = $\lambda xy.x$
din cele două alternative o alege pe prima
- F** = $\lambda xy.y$
din cele două alternative o alege pe a doua

Negăția

- $\mathbf{T} = \lambda xy.x$
- $\mathbf{F} = \lambda xy.y$
- $\mathbf{not} = \lambda b.b \mathbf{F} \mathbf{T}$

Negăția

- $\mathbf{T} = \lambda xy.x$
- $\mathbf{F} = \lambda xy.y$
- $\mathbf{not} = \lambda b.b \mathbf{F} \mathbf{T}$

Avem urmatoarele β -reducții:

- $\mathbf{not} \mathbf{T} = (\lambda b.b \mathbf{F} \mathbf{T})\mathbf{T} \rightarrow_{\beta} \mathbf{T} \mathbf{F} \mathbf{T} = (\lambda xy.x) \mathbf{F} \mathbf{T} \rightarrow_{\beta} \mathbf{F}$

Negăția

- $\mathbf{T} = \lambda xy.x$
- $\mathbf{F} = \lambda xy.y$
- $\mathbf{not} = \lambda b.b \mathbf{FT}$

Avem urmatoarele β -reducții:

- $\mathbf{not T} = (\lambda b.b \mathbf{FT})\mathbf{T} \rightarrow_{\beta} \mathbf{FTT} = (\lambda xy.x) \mathbf{FT} \rightarrow_{\beta} \mathbf{F}$
- $\mathbf{not F} = (\lambda b.b \mathbf{FT})\mathbf{F} \rightarrow_{\beta} \mathbf{FFT} = (\lambda xy.y) \mathbf{FT} \rightarrow_{\beta} \mathbf{T}$

Conjunctia

- **T** = $\lambda xy.x$
- **F** = $\lambda xy.y$
- **not** = $\lambda b.b \mathbf{F} \mathbf{T}$
- **and** = $\lambda ab.ab\mathbf{F}$

Conjunctia

- $\mathbf{T} = \lambda xy.x$
- $\mathbf{F} = \lambda xy.y$
- $\mathbf{not} = \lambda b.b \mathbf{F T}$
- $\mathbf{and} = \lambda ab.ab\mathbf{F}$

Avem urmatoarele β -reductii:

$$\square \mathbf{and} \mathbf{T T} = (\lambda ab.ab\mathbf{F}) \mathbf{T T} \rightarrow_{\beta} \mathbf{T T F} = (\lambda xy.x)\mathbf{T F} \rightarrow_{\beta} \mathbf{T}$$

Conjunctia

- $\mathbf{T} = \lambda xy.x$
- $\mathbf{F} = \lambda xy.y$
- $\mathbf{not} = \lambda b.b \mathbf{F T}$
- $\mathbf{and} = \lambda ab.ab\mathbf{F}$

Avem urmatoarele β -reductii:

- $\mathbf{and} \mathbf{T T} = (\lambda ab.ab\mathbf{F}) \mathbf{T T} \rightarrow_{\beta} \mathbf{T T F} = (\lambda xy.x)\mathbf{T F} \rightarrow_{\beta} \mathbf{T}$
- $\mathbf{and} \mathbf{T F} = (\lambda ab.ab\mathbf{F}) \mathbf{T F} \rightarrow_{\beta} \mathbf{T F F} = (\lambda xy.x)\mathbf{F F} \rightarrow_{\beta} \mathbf{F}$

Conjunctia

- $\mathbf{T} = \lambda xy.x$
- $\mathbf{F} = \lambda xy.y$
- $\mathbf{not} = \lambda b.b \mathbf{F T}$
- $\mathbf{and} = \lambda ab.ab\mathbf{F}$

Avem urmatoarele β -reductii:

- $\mathbf{and} \mathbf{T T} = (\lambda ab.ab\mathbf{F}) \mathbf{T T} \rightarrow_{\beta} \mathbf{T T F} = (\lambda xy.x)\mathbf{T F} \rightarrow_{\beta} \mathbf{T}$
- $\mathbf{and} \mathbf{T F} = (\lambda ab.ab\mathbf{F}) \mathbf{T F} \rightarrow_{\beta} \mathbf{T F F} = (\lambda xy.x)\mathbf{F F} \rightarrow_{\beta} \mathbf{F}$
- $\mathbf{and} \mathbf{F T} = (\lambda ab.ab\mathbf{F}) \mathbf{F T} \rightarrow_{\beta} \mathbf{F T F} = (\lambda xy.y)\mathbf{T F} \rightarrow_{\beta} \mathbf{F}$

Conjunctia

- $\mathbf{T} = \lambda xy.x$
- $\mathbf{F} = \lambda xy.y$
- $\mathbf{not} = \lambda b.b \mathbf{F T}$
- $\mathbf{and} = \lambda ab.ab\mathbf{F}$

Avem urmatoarele β -reductii:

- $\mathbf{and} \mathbf{T T} = (\lambda ab.ab\mathbf{F}) \mathbf{T T} \rightarrow_{\beta} \mathbf{T T F} = (\lambda xy.x)\mathbf{T F} \rightarrow_{\beta} \mathbf{T}$
- $\mathbf{and} \mathbf{T F} = (\lambda ab.ab\mathbf{F}) \mathbf{T F} \rightarrow_{\beta} \mathbf{T F F} = (\lambda xy.x)\mathbf{F F} \rightarrow_{\beta} \mathbf{F}$
- $\mathbf{and} \mathbf{F T} = (\lambda ab.ab\mathbf{F}) \mathbf{F T} \rightarrow_{\beta} \mathbf{F T F} = (\lambda xy.y)\mathbf{T F} \rightarrow_{\beta} \mathbf{F}$
- $\mathbf{and} \mathbf{F F} = (\lambda ab.ab\mathbf{F}) \mathbf{F F} \rightarrow_{\beta} \mathbf{F F F} = (\lambda xy.y)\mathbf{F F} \rightarrow_{\beta} \mathbf{F}$

Disjunctia

- **T** = $\lambda xy.x$
- **F** = $\lambda xy.y$
- **not** = $\lambda b.b \mathbf{F} \mathbf{T}$
- **and** = $\lambda ab.ab\mathbf{F}$
- **or** =

Disjunctia

- $\mathbf{T} = \lambda xy.x$
- $\mathbf{F} = \lambda xy.y$
- $\mathbf{not} = \lambda b.b \mathbf{F} \mathbf{T}$
- $\mathbf{and} = \lambda ab.ab\mathbf{F}$
- $\mathbf{or} = \lambda ab.a\mathbf{T}b$

Disjunctia

- $\mathbf{T} = \lambda xy.x$
- $\mathbf{F} = \lambda xy.y$
- $\mathbf{not} = \lambda b.b \mathbf{F} \mathbf{T}$
- $\mathbf{and} = \lambda ab.ab\mathbf{F}$
- $\mathbf{or} = \lambda ab.a\mathbf{T}b$

Avem urmatoarele β -reducții:

- $\mathbf{or} \mathbf{T} \mathbf{T} = (\lambda ab.a\mathbf{T}b) \mathbf{T} \mathbf{T} \rightarrow_{\beta} \mathbf{T} \mathbf{T} \mathbf{T} = (\lambda xy.x)\mathbf{T} \mathbf{T} \rightarrow_{\beta} \mathbf{T}$
- $\mathbf{or} \mathbf{T} \mathbf{F} = (\lambda ab.a\mathbf{T}b) \mathbf{T} \mathbf{F} \rightarrow_{\beta} \mathbf{T} \mathbf{T} \mathbf{F} = (\lambda xy.x)\mathbf{T} \mathbf{F} \rightarrow_{\beta} \mathbf{T}$
- $\mathbf{or} \mathbf{F} \mathbf{T} = (\lambda ab.a\mathbf{T}b) \mathbf{F} \mathbf{T} \rightarrow_{\beta} \mathbf{F} \mathbf{T} \mathbf{T} = (\lambda xy.y)\mathbf{T} \mathbf{T} \rightarrow_{\beta} \mathbf{T}$
- $\mathbf{or} \mathbf{F} \mathbf{F} = (\lambda ab.a\mathbf{T}b) \mathbf{F} \mathbf{F} \rightarrow_{\beta} \mathbf{F} \mathbf{T} \mathbf{F} = (\lambda xy.y)\mathbf{T} \mathbf{F} \rightarrow_{\beta} \mathbf{F}$

Conditional

Putem sa definim

- if $\mathbf{T} u w = u$**
- if $\mathbf{F} u w = w$**

Observati ca u si w sunt orice λ -termeni, nu doar Booleeni.

Conditional

Putem sa definim

- if** $T \ u \ w = u$
- if** $F \ u \ w = w$

Observati ca u si w sunt orice λ -termeni, nu doar Booleeni.

Avem

$$\mathbf{if} = \lambda b u w. b u w$$

unde b este un boolean.

Numere naturale

Intuiție: Capabilitatea de a itera o funcție de un număr de ori peste o valoare inițială

Codificare: Un număr natural este o funcție cu 2 argumente

- s** funcția care se iterează
- z** valoarea inițială

0 ::= $\lambda s z. z$ — s se iterează de 0 ori, deci valoarea inițială

Numere naturale

Intuiție: Capabilitatea de a itera o funcție de un număr de ori peste o valoare inițială

Codificare: Un număr natural este o funcție cu 2 argumente

- s** funcția care se iterează
- z** valoarea inițială

0 ::= $\lambda s z. z$ — s se iterează de 0 ori, deci valoarea inițială

1 ::= $\lambda s z. s z$ — funcția iterată o dată aplicată valorii initiale

Numere naturale

Intuiție: Capabilitatea de a itera o funcție de un număr de ori peste o valoare inițială

Codificare: Un număr natural este o funcție cu 2 argumente

- s funcția care se iterează
- z valoarea inițială

0 ::= $\lambda s z. z$ — s se iterează de 0 ori, deci valoarea inițială

1 ::= $\lambda s z. s z$ — funcția iterată o dată aplicată valorii inițiale

2 ::= $\lambda s z. s(s z)$ — s iterată de 2 ori, aplicată valorii inițiale

Numere naturale

Intuiție: Capabilitatea de a itera o funcție de un număr de ori peste o valoare inițială

Codificare: Un număr natural este o funcție cu 2 argumente

- s funcția care se iterează
- z valoarea inițială

0 ::= $\lambda s z. z$ — s se iterează de 0 ori, deci valoarea inițială

1 ::= $\lambda s z. s z$ — funcția iterată o dată aplicată valorii inițiale

2 ::= $\lambda s z. s(s z)$ — s iterată de 2 ori, aplicată valorii inițiale

...

8 ::= $\lambda s z. s(s(s(s(s(s(s(s(s z))))))))$

...

Numere naturale

Intuiție: Capabilitatea de a itera o funcție de un număr de ori peste o valoare inițială

Codificare: Un număr natural este o funcție cu 2 argumente

- s funcția care se iterează
- z valoarea inițială

0 ::= $\lambda s z. z$ — s se iterează de 0 ori, deci valoarea inițială

1 ::= $\lambda s z. s z$ — funcția iterată o dată aplicată valorii inițiale

2 ::= $\lambda s z. s(s z)$ — s iterată de 2 ori, aplicată valorii inițiale

...

8 ::= $\lambda s z. s(s(s(s(s(s(s(s(s z))))))))$

...

Observație: **0 = false**

Operații aritmetice de bază

succ ::= $\lambda n\ s\ z.s(n\ s\ z)$

Operații aritmetice de bază

succ ::= $\lambda n s z.s(n s z)$

succ 0 = $(\lambda n s z.s(n s z))0 \rightarrow_{\beta} \lambda s z.s(0 s z) \rightarrow_{\beta} \lambda s z.s z = 1$

Operații aritmetice de bază

succ ::= $\lambda n s z.s(n s z)$

succ 0 = $(\lambda n s z.s(n s z))0 \rightarrow_{\beta} \lambda s z.s(0 s z) \rightarrow_{\beta} \lambda s z.s z = 1$

plus ::= $\lambda m n s z.m s(n s z)$

Operații aritmetice de bază

succ ::= $\lambda n s z. s(n s z)$

succ 0 = $(\lambda n s z. s(n s z)) \mathbf{0} \rightarrow_{\beta} \lambda s z. s(\mathbf{0} s z) \rightarrow_{\beta} \lambda s z. s z = \mathbf{1}$

plus ::= $\lambda m n s z. m s(n s z)$

plus 3 2 = $(\lambda m n s z. m s(n s z)) \mathbf{3} \mathbf{2} \rightarrow_{\beta} \lambda s z. \mathbf{3} s(\mathbf{2} s z)$
 $\rightarrow_{\beta} \lambda s z. s(s(s(\mathbf{2} s z))) \rightarrow_{\beta} \lambda s z. s(s(s(s(s(z)))))) = \mathbf{5}$

Operații aritmetice de bază

succ ::= $\lambda n s z.s(n s z)$

$$\text{succ } \mathbf{0} = (\lambda n s z.s(n s z)) \mathbf{0} \rightarrow_{\beta} \lambda s z.s(\mathbf{0} s z) \rightarrow_{\beta} \lambda s z.s z = \mathbf{1}$$

plus ::= $\lambda m n s z.m s(n s z)$

$$\begin{aligned}\text{plus } \mathbf{3} \ \mathbf{2} &= (\lambda m n s z.m s(n s z)) \mathbf{3} \ \mathbf{2} \rightarrow_{\beta} \lambda s z.\mathbf{3} s(\mathbf{2} s z) \\ &\rightarrow_{\beta} \lambda s z.s(s(s(\mathbf{2} s z))) \rightarrow_{\beta} \lambda s z.s(s(s(s(s(z)))))) = \mathbf{5}\end{aligned}$$

mult ::= $\lambda m n s.m(n s)$

Scaderea este mai complicata deoarece nu exista numere negative.

Perechi

Intuiție: Capabilitatea de a aplica o funcție componentelor perechii

Codificare: O funcție cu 3 argumente

reprezentând componentele perechii și funcția ce vrem să o aplicăm lor.

pair ::= $\lambda x y. \lambda f. f x y$

Constructorul de perechi

Exemplu: **pair** $x y \rightarrow_{\beta}^2 \lambda f. f x y$

perechea (x, y) reprezintă capabilitatea de a aplica o funcție de două argumente lui x și apoi lui y .

Operații pe perechi

pair ::= $\lambda x y. \lambda f. f x y$

pair $xy \equiv_{\beta} f x y$

fst ::= $\lambda p. p \text{ true} — \text{true alege prima componentă}$

$\text{fst} (\text{pair } x y) \rightarrow_{\beta} \text{pair } x y \text{ true} \rightarrow_{\beta}^3 \text{true } x y \rightarrow_{\beta}^2 x$

snd ::= $\lambda p. p \text{ false} — \text{false alege a doua componentă}$

$\text{snd} (\text{pair } x y) \rightarrow_{\beta} \text{pair } x y \text{ false} \rightarrow_{\beta}^3 \text{false } x y \rightarrow_{\beta}^2 y$

Liste

Intuiție: Capabilitatea de a adăuga o listă

Codificare: O funcție cu 2 argumente:

funcția de agregare și valoarea initială

Lista [3, 5] este reprezentată prin a 3 (a 5 i)

Liste

Intuiție: Capabilitatea de a agrega o listă

Codificare: O funcție cu 2 argumente:

funcția de agregare și valoarea initială

Lista [3, 5] este reprezentată prin a 3 (a 5 i)

null ::= $\lambda a. i.i$ — lista vidă

cons ::= $\lambda x. l. \lambda a. i. a x (l a i)$

Constructorul de liste

Exemplu: $\text{cons } 3 (\text{cons } 5 \text{ null}) \xrightarrow{\beta}^2 \lambda a. i. a 3 (\text{cons } 5 \text{ null}$
 $a i) \xrightarrow{\beta}^4 \lambda a. i. a 3 (a 5 (\text{null } a i)) \xrightarrow{\beta}^2 \lambda a. i. a 3 (a 5 i)$

Lista [3, 5] reprezintă capabilitatea de a agrega elementele 3 și apoi 5 dată fiind o funcție de agregare și o valoare implicită *i*.

Pe săptămâna viitoare!

Curs 14

2021-2022

Fundamentele limbajelor de programare

Cuprins

1 Examen

2 Corespondența Curry-Howard-Lambek

Examen

Informatii generale despre examen

- 24 iunie
- 2 ore, fizic, în laboratoare/amfiteatre
- cu materialele ajutătoare de la curs/seminar/laborator
- 1 punct din oficiu
- condiția minimă pentru a promova: nota examen > 4.99

Informatii generale despre examen

- Trebuie să rezolvați atât probleme pe calculator, cât și pe foaie
- Dacă dorîți să susțineți examenul pe laptop-ul personal, vă rugăm să completați formularul de mai jos (primul venit, primul servit, în funcție de numărul de locuri disponibile):
 - seria 23:**
<https://tinyurl.com/4t4y74aa>
 - seria 24:**
<https://tinyurl.com/5tu5x2h7>
 - seria 25:**
<https://tinyurl.com/3ymmm9hsz>
- Termen limită pentru completarea formularelor: **2.06.2022**

Structură examen

□ Partea teoretică (4 puncte) - 3 probleme din lista de mai jos:

- unificare
- deducție naturală
- puncte fixe
- rezoluție
- arbori de execuție și arbori SLD
- pași în semantica operațională
- substituții și β -reduceri în lambda calcul

□ Partea practică (5 puncte)

- 1** o problemă tipică de Prolog (2 puncte)
- 2** o problemă de limbaj de programare (3 puncte)
 - se dă sintaxa unui limbaj de programare
 - să se verifice dacă un sir de caractere este un program corect în limbaj
 - să se implementeze un interpretor pentru limbaj

Coresponden  Curry-Howard-Lambek

Schimbați perspectiva



Roger Antonsen
Universitatea din Oslo

TED Talk: Math is the hidden secret to understanding the world

... înțelegerea este legată de abilitatea de a-ți schimba perspectiva.

https://www.ted.com/talks/roger_antonsen_math_is_the_hidden_secret_to_understanding_the_world

Corespondența Curry-Howard-Lambek

Ne vom uita la niște concepte din trei perspective diferite:

- Teoria Tipurilor**
- Logică**
- Teoria Categoriilor**

Un program simplu în Haskell

```
data Point = Point Int Int
```

```
makePoint :: Int -> Int -> Point  
makePoint x y = Point x y
```

```
getX :: Point -> Int  
getX (Point x y) = x
```

```
getY :: Point -> Int  
getY (Point x y) = y
```

```
origin :: Point  
origin = makePoint 0 0
```

Un program simplu în Haskell

Hai să schimbăm perspectiva!

Un program simplu în Haskell

Hai să schimbăm perspectiva!

```
data Point = Point Int Int
```

```
makePoint :: Int -> Int -> Point  
makePoint x y = Point x y
```

$$\frac{x : \text{Int} \quad y : \text{Int}}{\text{makePoint } x \text{ } y : \text{Point}}$$

Un program simplu în Haskell

Hai să schimbăm perspectiva!

```
data Point = Point Int Int
```

```
makePoint :: Int -> Int -> Point  
makePoint x y = Point x y
```

```
getX :: Point -> Int  
getX (Point x y) = x
```

$$\frac{x : \text{Int} \quad y : \text{Int}}{\text{makePoint } x \text{ } y : \text{Point}}$$

$$\frac{p : \text{Point}}{\text{getX } p : \text{Int}}$$

Un program simplu în Haskell

Hai să schimbăm perspectiva!

```
data Point = Point Int Int
```

```
makePoint :: Int -> Int -> Point  
makePoint x y = Point x y
```

```
getX :: Point -> Int  
getX (Point x y) = x
```

```
getY :: Point -> Int  
getY (Point x y) = y
```

$$\frac{x : \text{Int} \quad y : \text{Int}}{\text{makePoint } x \text{ } y : \text{Point}}$$

$$\frac{p : \text{Point}}{\text{getX } p : \text{Int}}$$

$$\frac{p : \text{Point}}{\text{getY } p : \text{Int}}$$

Un program simplu în Haskell

Hai să schimbăm perspectiva!

```
data Point = Point Int Int
```

```
makePoint :: Int -> Int -> Point  
makePoint x y = Point x y
```

```
getX :: Point -> Int  
getX (Point x y) = x
```

```
getY :: Point -> Int  
getY (Point x y) = y
```

$$\frac{x : \text{Int} \quad y : \text{Int}}{\text{makePoint } x y : \text{Point}} (\text{Point}_I)$$

$$\frac{p : \text{Point}}{\text{getX } p : \text{Int}} (\text{Point}_{E_1})$$

$$\frac{p : \text{Point}}{\text{getY } p : \text{Int}} (\text{Point}_{E_2})$$

Generalizzare

$$\frac{x : \text{Int} \quad y : \text{Int}}{\text{makePoint } x \ y : \text{Point}_I} \ (\text{Point}_I)$$

$$\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B} \ (\times_I)$$

Generalizzare

$$\frac{x : \text{Int} \quad y : \text{Int}}{\text{makePoint } x \ y : \text{Point}} \ (\text{Point}_I)$$

$$\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B} \ (\times_I)$$

$$\frac{p : \text{Point}}{\text{getX } p : \text{Int}} \ (\text{Point}_{E_1})$$

$$\frac{p : A \times B}{\text{fst } p : A} \ (\times_{E_1})$$

Generalizzare

$$\frac{x : \text{Int} \quad y : \text{Int}}{\text{makePoint } x \ y : \text{Point}} \ (\text{Point}_I)$$

$$\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B} \ (\times_I)$$

$$\frac{p : \text{Point}}{\text{getX } p : \text{Int}} \ (\text{Point}_{E_1})$$

$$\frac{p : A \times B}{\text{fst } p : A} \ (\times_{E_1})$$

$$\frac{p : \text{Point}}{\text{getY } p : \text{Int}} \ (\text{Point}_{E_2})$$

$$\frac{p : A \times B}{\text{snd } p : B} \ (\times_{E_2})$$

Alt exemplu simplu

```
> let f = (\x -> x * 3) :: Int -> Int
```

Alt exemplu simplu

```
> let f = (\x -> x * 3) :: Int -> Int
```

$$\frac{[x : \text{Int}] \quad ; \quad x * 3 : \text{Int}}{\lambda x. x * 3 : \text{Int} \rightarrow \text{Int}}$$

Alt exemplu simplu

```
> let f = (\x -> x * 3) :: Int -> Int
```

$$\frac{x : \text{Int} \quad | \quad x * 3 : \text{Int}}{\lambda x. x * 3 : \text{Int} \rightarrow \text{Int}}$$

```
> f 5  
15
```

Alt exemplu simplu

```
> let f = (\x -> x * 3) :: Int -> Int
```

$$\frac{\begin{array}{c} [x : \text{Int}] \\ \vdots \\ x * 3 : \text{Int} \end{array}}{\lambda x. x * 3 : \text{Int} \rightarrow \text{Int}}$$


```
> f 5
```

$$\frac{f : \text{Int} \rightarrow \text{Int} \quad 5 : \text{Int}}{f 5 : \text{Int}}$$

Alt exemplu simplu

```
> let f = (\x -> x * 3) :: Int -> Int
```

$[x : \text{Int}]$

$$\frac{x * 3 : \text{Int}}{\lambda x. x * 3 : \text{Int} \rightarrow \text{Int}} (\text{fun}_I)$$


```
> f 5
```

$$\frac{f : \text{Int} \rightarrow \text{Int} \quad 5 : \text{Int}}{f 5 : \text{Int}} (\text{fun}_E)$$

Generalizzare

$$\frac{[\textcolor{teal}{x} : \textcolor{violet}{Int}] \quad \vdots \quad [\textcolor{teal}{x} * 3 : \textcolor{violet}{Int}]}{\lambda x. x * 3 : \textcolor{violet}{Int} \rightarrow \textcolor{violet}{Int}} \ (\textit{fun}_I)$$
$$\frac{[\textcolor{teal}{x} : \textcolor{violet}{A}] \quad \vdots \quad \textcolor{teal}{b} : \textcolor{violet}{B}}{\lambda x. \textcolor{teal}{b} : \textcolor{violet}{A} \rightarrow \textcolor{violet}{B}} \ (\rightarrow_I)$$

Generalizzare

$$\frac{[\textcolor{teal}{x} : \textcolor{violet}{Int}] \quad \vdots \quad \textcolor{teal}{x} * 3 : \textcolor{violet}{Int}}{\lambda x. x * 3 : \textcolor{violet}{Int} \rightarrow \textcolor{violet}{Int}} \ (\textit{fun}_I)$$

$$\frac{[\textcolor{teal}{x} : \textcolor{violet}{A}] \quad \vdots \quad \textcolor{teal}{b} : \textcolor{violet}{B}}{\lambda x. \textcolor{teal}{b} : \textcolor{violet}{A} \rightarrow \textcolor{violet}{B}} \ (\rightarrow_I)$$

$$\frac{\textcolor{teal}{f} : \textcolor{violet}{Int} \rightarrow \textcolor{violet}{Int} \quad \textcolor{teal}{f} \ 5 : \textcolor{violet}{Int}}{\textcolor{teal}{f} \ 5 : \textcolor{violet}{Int}} \ (\textit{fun}_E)$$

$$\frac{\textcolor{teal}{f} : \textcolor{violet}{A} \rightarrow \textcolor{violet}{B} \quad \textcolor{teal}{x} : \textcolor{violet}{A}}{\textcolor{teal}{f} \ x : \textcolor{violet}{B}} \ (\rightarrow_E)$$

Generalizare

$$\frac{[\textcolor{teal}{x} : \textcolor{violet}{Int}] \quad \vdots \quad \textcolor{teal}{x} * 3 : \textcolor{violet}{Int}}{\lambda x. x * 3 : \textcolor{violet}{Int} \rightarrow \textcolor{violet}{Int}} \ (\textit{fun}_I)$$

$$\frac{[\textcolor{teal}{x} : \textcolor{violet}{A}] \quad \vdots \quad \textcolor{teal}{b} : \textcolor{violet}{B}}{\lambda x. \textcolor{teal}{b} : \textcolor{violet}{A} \rightarrow \textcolor{violet}{B}} \ (\rightarrow_I)$$

(abstracție)

$$\frac{f : \textcolor{violet}{Int} \rightarrow \textcolor{violet}{Int} \quad 5 : \textcolor{violet}{Int}}{f 5 : \textcolor{violet}{Int}} \ (\textit{fun}_E)$$

$$\frac{f : \textcolor{violet}{A} \rightarrow \textcolor{violet}{B} \quad x : \textcolor{violet}{A}}{f x : \textcolor{violet}{B}} \ (\rightarrow_E)$$

(aplicare)

Un λ -calcul cu tipuri

$$\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B} (\times_I)$$

$$\frac{p : A \times B}{fst\ p : A} (\times_{E_1})$$

$$\frac{p : A \times B}{snd\ p : B} (\times_{E_2})$$

$$[x : A]$$

⋮

$$\frac{b : B}{\lambda x.b : A \rightarrow B} (\rightarrow_I)$$

$$\frac{f : A \rightarrow B \quad x : A}{f\ x : B} (\rightarrow_E)$$

Logica. Ce este adevărt și ce este fals?

Dacă afară este întuneric atunci, dacă porcii zboară atunci este întuneric afară.

Logica. Ce este adevărt și ce este fals?

Dacă afară este întuneric atunci, dacă porcii zboară atunci este întuneric afară.

$$\begin{array}{lcl} A & = & \text{afară este întuneric} \\ B & = & \text{porcii zboară} \end{array} \qquad A \supset (B \supset A)$$

Logica. Ce este adevărt și ce este fals?

Dacă afară este întuneric atunci, dacă porcii zboară atunci este întuneric afară.

$$\begin{array}{lcl} A & = & \text{afară este întuneric} \\ B & = & \text{porcii zboară} \end{array} \qquad A \supset (B \supset A)$$

Este adevărată această afirmație?

Logica. Ce este adevărt și ce este fals?

Dacă afară este întuneric atunci, dacă porcii zboară atunci este întuneric afară.

$$\begin{array}{lcl} A & = & \text{afară este întuneric} \\ B & = & \text{porcii zboară} \end{array} \qquad A \supset (B \supset A)$$

Este adevărată această afirmație?

A	B	$B \supset A$	$A \supset (B \supset A)$
false	false	true	true
false	true	false	true
true	false	true	true
true	true	true	true

Logica. Ce este adevărt și ce este fals?

Dacă afară este întuneric atunci, dacă porcii zboară atunci este întuneric afară.

$$\begin{array}{lcl} A & = & \text{afară este întuneric} \\ B & = & \text{porcii zboară} \end{array} \qquad A \supset (B \supset A)$$

Este adevărată această afirmație?

A	B	$B \supset A$	$A \supset (B \supset A)$
false	false	true	true
false	true	false	true
true	false	true	true
true	true	true	true

Da!

Semantica

Dăm valori atomilor în mulțimea $\{0, 1\}$, definim o evaluare $e : Atoms \rightarrow \{0, 1\}$.

Semantica

Dăm valori atomilor în mulțimea $\{0, 1\}$, definim o evaluare $e : Atoms \rightarrow \{0, 1\}$.

Având o evaluare, putem să o extindem la formule folosind tabelele de adevăr:

$$\& : \{0, 1\} \times \{0, 1\} \rightarrow \{0, 1\} \quad \supset : \{0, 1\} \times \{0, 1\} \rightarrow \{0, 1\}$$

A	B	$A \& B$
0	0	0
0	1	0
1	0	0
1	1	1

A	B	$A \supset B$
0	0	1
0	1	1
1	0	0
1	1	1

Semantica

Dăm valori atomilor în mulțimea $\{0, 1\}$, definim o evaluare $e : Atoms \rightarrow \{0, 1\}$.

Având o evaluare, putem să o extindem la formule folosind tabelele de adevăr:

$$\& : \{0, 1\} \times \{0, 1\} \rightarrow \{0, 1\} \quad \supset : \{0, 1\} \times \{0, 1\} \rightarrow \{0, 1\}$$

A	B	$A \& B$
0	0	0
0	1	0
1	0	0
1	1	1

A	B	$A \supset B$
0	0	1
0	1	1
1	0	0
1	1	1

Dacă pentru toate evaluările posibile, o formulă are valoarea de adevăr 1, atunci spunem că este mereu adevărată (este o **tautologie**).

Sintaxa unei logici

- Noțiunile de teoremă și demonstrabilitate
- Oferă metode de a manipula simboluri din logică (i.e., atomi, \supset , $\&$) pentru a stabili când o formulă este demonstrabilă (aka este teoremă).

Sintaxa unei logici

- Noțiunile de teoremă și demonstrabilitate
- Oferă metode de a manipula simboluri din logică (i.e., atomi, \supset , $\&$) pentru a stabili când o formulă este demonstrabilă (aka este teoremă).

Completitudine = sintaxa și semantica coincid

Corectitudine = sintaxa implică semantica

Un sistem de deducție naturală

- Reguli pentru a manevra fiecare conector logic
- Reguli pentru introducerea si eliminarea conectorilor
- Reguli de forma

Ipoteze
Concluzie

Un sistem de deducție naturală

- Reguli pentru a manevra fiecare conector logic
- Reguli pentru introducerea si eliminarea conectorilor
- Reguli de forma

Ipoteze
Concluzie

$$\frac{A \quad B}{A \& B} (\&_I)$$

Un sistem de deducție naturală

- Reguli pentru a manevra fiecare conector logic
- Reguli pentru introducerea si eliminarea conectorilor
- Reguli de forma

Ipoteze

Concluzie

$$\frac{A \quad B}{A \& B} (\&_I)$$

$$\frac{A \& B}{A} (\&_{E_1})$$

$$\frac{A \& B}{B} (\&_{E_2})$$

Un sistem de deducție naturală

- Reguli pentru a manevra fiecare conector logic
- Reguli pentru introducerea si eliminarea conectorilor
- Reguli de forma

Ipoteze
Concluzie

$$\frac{A \quad B}{A \& B} (\&_I)$$

$$\frac{A \& B}{A} (\&_{E_1})$$

$$\frac{A \& B}{B} (\&_{E_2})$$

[A]

⋮

$$\frac{B}{A \supset B} (\supset_I)$$

Un sistem de deducție naturală

- Reguli pentru a manevra fiecare conector logic
- Reguli pentru introducerea și eliminarea conectorilor
- Reguli de formă

Ipoteze
Concluzie

$$\frac{A \quad B}{A \& B} (\&_I)$$

$$\frac{A \& B}{A} (\&_{E_1})$$

$$\frac{A \& B}{B} (\&_{E_2})$$

[A]

⋮

B

$$\frac{}{A \supset B} (\supset_I)$$

$$\frac{A \supset B \quad A}{B} (\supset_E)$$

Un sistem de deducție naturală

- Reguli pentru a manevra fiecare conector logic
- Reguli pentru introducerea și eliminarea conectorilor
- Reguli de formă

Ipoteze
Concluzie

$$\frac{A \quad B}{A \& B} (\&_I)$$

$$\frac{A \& B}{A} (\&_{E_1})$$

$$\frac{A \& B}{B} (\&_{E_2})$$

[A]

⋮

B

$$\frac{}{A \supset B} (\supset_I)$$

$$\frac{A \supset B \quad A}{B} (\supset_E)$$

Arată cunoscut?

Ce am văzut până acum

Un λ -calcul cu tipuri

$$\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B} (\times_I)$$

$$\frac{p : A \times B}{fst\ p : A} (\times_{E_1})$$

$$\frac{p : A \times B}{snd\ p : B} (\times_{E_2})$$

$$[x : A]$$

⋮

$$b : B$$

$$\frac{}{\lambda x. n : A \rightarrow B} (\rightarrow_I)$$

$$\frac{f : A \rightarrow B \quad x : A}{f x : B} (\rightarrow_E)$$

Un sistem de deducție naturală

$$\frac{A \quad B}{A \& B} (\&_I)$$

$$\frac{A \& B}{A} (\&_{E_1})$$

$$\frac{A \& B}{B} (\&_{E_2})$$

$$[A]$$

⋮

$$B$$

$$\frac{}{A \supset B} (\supset_I)$$

$$\frac{A \supset B \quad A}{B} (\supset_E)$$

Ce am văzut până acum

Un λ -calcul cu tipuri

$$\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B} (\times_I)$$

$$\frac{p : A \times B}{fst \ p : A} (\times_{E_1})$$

$$\frac{p : A \times B}{snd \ p : B} (\times_{E_2})$$

$$[x : A]$$

⋮

$$b : B$$

$$\frac{}{\lambda x. n : A \rightarrow B} (\rightarrow_I)$$

$$\frac{f : A \rightarrow B \quad x : A}{f x : B} (\rightarrow_E)$$

Un sistem de deducție naturală

$$\frac{A \quad B}{A \& B} (\&_I)$$

$$\frac{A \& B}{A} (\&_{E_1})$$

$$\frac{A \& B}{B} (\&_{E_2})$$

$$[A]$$

⋮

$$B$$

$$\frac{}{A \supset B} (\supset_I)$$

$$\frac{A \supset B \quad A}{B} (\supset_E)$$

Propositions are types! ♡

Să analizăm mai atent

Un λ -calcul cu tipuri Un sistem de deducție naturală

$$\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B} (\times_I)$$

$$\frac{A \quad B}{A \& B} (\&_I)$$

Să analizăm mai atent

Un λ -calcul cu tipuri Un sistem de deducție naturală

$$\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B} (\times_I)$$

$$\frac{A \quad B}{A \& B} (\&_I)$$

Faptul că există un termen de tip A (*inhabitation of type A*)
înseamnă că A este teoremă în logică! ♡

Să analizăm mai atent

Un λ -calcul cu tipuri

Un sistem de deducție naturală

$$: A \rightarrow A$$

Să analizăm mai atent

Un λ -calcul cu tipuri

Un sistem de deducție naturală

$$\lambda x.x : A \rightarrow A \quad (\text{id})$$

Să analizăm mai atent

Un λ -calcul cu tipuri

$$\lambda x.x : A \rightarrow A \quad (\text{id})$$

Un sistem de deducție naturală

$$\overline{A \supset A}$$

Să analizăm mai atent

Un λ -calcul cu tipuri

$$\lambda x.x : A \rightarrow A \quad (\text{id})$$

Un sistem de deducție naturală

$$[A]$$

$$\overline{A \supset A}$$

Să analizăm mai atent

Un λ -calcul cu tipuri

$$\lambda x.x : A \rightarrow A \quad (\text{id})$$

Un sistem de deducție naturală

$$\frac{[A] \quad A}{A \supset A}$$

Să analizăm mai atent

Un λ -calcul cu tipuri

$$\lambda x.x : A \rightarrow A \quad (\text{id})$$

Un sistem de deducție naturală

$$\frac{[A]}{A}$$

$$: A \rightarrow (B \rightarrow A)$$

Să analizăm mai atent

Un λ -calcul cu tipuri

$$\lambda x.x : A \rightarrow A \quad (\text{id})$$

Un sistem de deducție naturală

$$\frac{[A]}{A}$$

$$\lambda x.(\lambda y.x) : A \rightarrow (B \rightarrow A) \quad (\text{const})$$

Să analizăm mai atent

Un λ -calcul cu tipuri

$$\lambda x.x : A \rightarrow A \quad (\text{id})$$

Un sistem de deducție naturală

$$\frac{[A]}{A}$$

$$\lambda x.(\lambda y.x) : A \rightarrow (B \rightarrow A) \quad (\text{const})$$

$$\overline{A \supset (B \supset A)}$$

Să analizăm mai atent

Un λ -calcul cu tipuri

$$\lambda x.x : A \rightarrow A \quad (\text{id})$$

Un sistem de deducție naturală

$$\frac{[A]}{A}$$

$$[A]$$

$$\lambda x.(\lambda y.x) : A \rightarrow (B \rightarrow A) \quad (\text{const})$$

$$\frac{B \supset A}{A \supset (B \supset A)}$$

Să analizăm mai atent

Un λ -calcul cu tipuri

$$\lambda x.x : A \rightarrow A \quad (\text{id})$$

Un sistem de deducție naturală

$$\frac{[A]}{A}$$

$$\lambda x.(\lambda y.x) : A \rightarrow (B \rightarrow A) \quad (\text{const})$$

$$\frac{\overline{[A] \\ [B]}}{A \supset (B \supset A)}$$

Să analizăm mai atent

Un λ -calcul cu tipuri

$$\lambda x.x : A \rightarrow A \quad (\text{id})$$

Un sistem de deducție naturală

$$\frac{[A] \quad A}{A \supset A}$$

$$\lambda x.(\lambda y.x) : A \rightarrow (B \rightarrow A) \quad (\text{const})$$

$$\frac{[A] \quad [B] \quad A}{B \supset A} \quad A \supset (B \supset A)$$

Să analizăm mai atent

Un λ -calcul cu tipuri

$$\lambda x.x : A \rightarrow A \quad (\text{id})$$

Un sistem de deducție naturală

$$\frac{[A]}{A}$$

$$\lambda x.(\lambda y.x) : A \rightarrow (B \rightarrow A) \quad (\text{const})$$

$$\frac{\begin{array}{c} [A] \\ [B] \\ A \end{array}}{B \supset A}$$

$$\frac{}{A \supset (B \supset A)}$$

Proofs are Terms! ❤

Demonstrațiile sunt termeni!

Corespondența Curry-Howard

Teoria Tipurilor	Logică
tipuri	formule
termeni	demonstrații
<i>inhabitation</i> a tipului A	demonstrație a lui A

Corespondența Curry-Howard

Teoria Tipurilor	Logică
tipuri	formule
termeni	demonstrații
<i>inhabitation</i> a tipului A	demonstrație a lui A
tip produs	conjuncție
tip funcție	implicație

Corespondența Curry-Howard

Teoria Tipurilor	Logică
tipuri	formule
termeni	demonstrații
<i>inhabitation</i> a tipului A	demonstrație a lui A
tip produs	conjuncție
tip funcție	implicație
tip sumă	disjuncție
tipul void	false
tipul unit	true

Deducre naturală pentru logica clasică

[A]

⋮

B

$\frac{}{A \supset B} (\supset_I)$

$\frac{A \supset B \quad A}{B} (\supset_E)$

$\frac{A \quad B}{A \& B} (\&_I)$

$\frac{A \& B}{A} (\&_{E_1})$

$\frac{A \& B}{B} (\&_{E_2})$

Deducre naturală pentru logica clasică

[A]

⋮

B

$$\frac{}{A \supset B} (\supset_I)$$

$$\frac{A \supset B \quad A}{B} (\supset_E)$$

$$\frac{A \quad B}{A \& B} (\&_I)$$

$$\frac{A \& B}{A} (\&_{E_1})$$

$$\frac{A \& B}{B} (\&_{E_2})$$

$$\frac{A}{A \vee B} (\vee_{I_1})$$

$$\frac{B}{A \vee B} (\vee_{I_2})$$

$$\frac{A \vee B \quad A \supset C \quad B \supset C}{C} (\vee_E)$$

Deducre naturală pentru logica clasică

[A]

⋮

B

$$\frac{}{A \supset B} (\supset_I)$$

$$\frac{A \supset B \quad A}{B} (\supset_E)$$

$$\frac{A \quad B}{A \& B} (\&_I)$$

$$\frac{A \& B}{A} (\&_{E_1})$$

$$\frac{A \& B}{B} (\&_{E_2})$$

$$\frac{A}{A \vee B} (\vee_{I_1})$$

$$\frac{B}{A \vee B} (\vee_{I_2})$$

$$\frac{A \vee B \quad A \supset C \quad B \supset C}{C} (\vee_E)$$

$$\frac{\perp}{A} (\text{ex falso quodlibet})$$

Deducre naturală pentru logica clasică

[A]

⋮

B

$\frac{B}{A \supset B}$ (\supset_I)

$\frac{A \supset B \quad A}{B}$ (\supset_E)

$\frac{A \quad B}{A \& B}$ ($\&_I$)

$\frac{A \& B}{A}$ ($\&_{E_1}$)

$\frac{A \& B}{B}$ ($\&_{E_2}$)

$\frac{A}{A \vee B}$ (\vee_{I_1})

$\frac{B}{A \vee B}$ (\vee_{I_2})

$\frac{A \vee B \quad A \supset C \quad B \supset C}{C}$ (\vee_E)

$\frac{\perp}{A}$ (ex falso quodlibet)

$\frac{\neg \neg A}{A}$ (reductio ad absurdum)

($\neg A$ este o abreviere pentru $A \supset \perp$)

Deducre naturală pentru logica intuiționistă

[A]

⋮

$$\frac{B}{A \supset B} (\supset_I)$$

$$\frac{A \supset B \quad A}{B} (\supset_E)$$

$$\frac{A \quad B}{A \& B} (\&_I)$$

$$\frac{A \& B}{A} (\&_{E_1})$$

$$\frac{A \& B}{B} (\&_{E_2})$$

$$\frac{A}{A \vee B} (\vee_{I_1})$$

$$\frac{B}{A \vee B} (\vee_{I_2})$$

$$\frac{A \vee B \quad A \supset C \quad B \supset C}{C} (\vee_E)$$

$$\frac{\perp}{A} (\text{ex falso quodlibet})$$

Logica intuiționistă

- Logică constructivistă
- Bazată pe noțiunea de demonstrație
- Utilă deoarece demonstrațiile sunt executabile și produc exemple
- Baza pentru *proof assistants* (e.g., Coq, Lean, Isabele, Agda, Idris)

Logica intuiționistă

- Logică constructivistă
- Bazată pe noțiunea de demonstrație
- Utilă deoarece demonstrațiile sunt executabile și produc exemple
- Baza pentru *proof assistants* (e.g., Coq, Lean, Isabele, Agda, Idris)
- Următoarele formule echivalente nu sunt demonstrabile în logica intuiționistă:
 - dubla negație: $\neg\neg A \supset A$
 - excluded middle: $A \vee \neg A$
 - legea lui Pierce: $((A \supset B) \supset A) \supset A$

Logica intuiționistă

- Logică constructivistă
- Bazată pe noțiunea de demonstrație
- Utilă deoarece demonstrațiile sunt executabile și produc exemple
- Baza pentru *proof assistants* (e.g., Coq, Lean, Isabele, Agda, Idris)
- Următoarele formule echivalente nu sunt demonstrabile în logica intuiționistă:
 - dubla negație: $\neg\neg A \supset A$
 - excluded middle: $A \vee \neg A$
 - legea lui Pierce: $((A \supset B) \supset A) \supset A$
- Nu există semantică cu tabele de adevăr pentru logica intuiționistă!
Are semantici alternative (e.g., semantica de tip Kripke)

Logica intuiționistă

- Logică constructivistă
- Bazată pe noțiunea de demonstrație
- Utilă deoarece demonstrațiile sunt executabile și produc exemple
- Baza pentru *proof assistants* (e.g., Coq, Lean, Isabele, Agda, Idris)
- Următoarele formule echivalente nu sunt demonstrabile în logica intuiționistă:
 - dubla negație: $\neg\neg A \supset A$
 - excluded middle: $A \vee \neg A$
 - legea lui Pierce: $((A \supset B) \supset A) \supset A$
- Nu există semantică cu tabele de adevăr pentru logica intuiționistă!
Are semantici alternative (e.g., semantica de tip Kripke)

Înțial, Corespondența Curry-Howard a fost între
Church's simply typed λ -calculus și Gentzen's natural deduction
for intuitionistic logic.

Corespondența Curry-Howard în general

Teoria Tipurilor	Logică
tipuri	formule
termeni	demonstrații
<i>inhabitation</i> a tipului A	demonstrație a lui A
tip produs	conjuncție
tip funcție	implicație
tip sumă	disjuncție
tipul void	false
tipul unit	true
dependent types	cuantificatori
call/cc operator	Peirce's law
monade	o logică modală
...	...

De ce?

- Este pur si simplu fascinant

De ce?

- Este pur și simplu fascinant
- Nu gândiți logica și informatica ca domenii diferite.

De ce?

- Este pur și simplu fascinant
- Nu gândiți logica și informatica ca domenii diferite.
- Gândind din perspective diferite ne poate ajuta să știm ce este posibil/imposibil.

De ce?

- Este pur și simplu fascinant
- Nu gândiți logica și informatica ca domenii diferite.
- Gândind din perspective diferite ne poate ajuta să știm ce este posibil/imposibil.
- Teoria tipurilor nu ar trebui să fie o adunătură *ad hoc* de reguli!

Să schimbăm iar perspectiva!

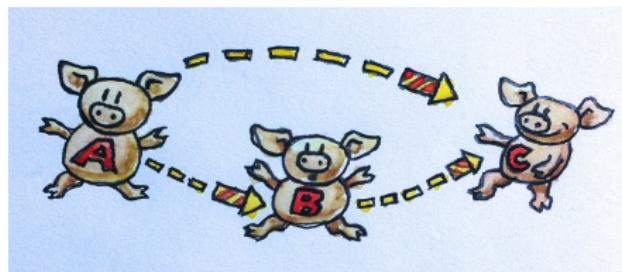
Teoria categoriilor

- A category is an embarrassingly simple concept.

Bartosz Milewski, Category Theory for Programmers

- Categorie = obiecte + săgeți

- Ingredient cheie: compunerea de săgeți



credits: Bartosz Milewski

O categorie

O **categorie** C constă în

- Obiecte:
- Săgeți:
- Compunere:

O categorie

O categorie **C** constă în

- Obiecte: notate A, B, C, \dots
- Săgeți:
- Compunere:

O categorie

O categorie **C** constă în

- **Obiecte:** notate A, B, C, \dots
- **Săgeți:** pentru orice obiecte A și B , există o mulțime de săgeți $\mathbf{C}(A, B)$
 - notăm $f \in \mathbf{C}(A, B)$ cu $f : A \rightarrow B$ sau $A \xrightarrow{f} B$
- **Compunere:**

O categorie

O categorie **C** constă în

- **Obiecte:** notate A, B, C, \dots
- **Săgeți:** pentru orice obiecte A și B , există o mulțime de săgeți $\mathbf{C}(A, B)$
 - notăm $f \in \mathbf{C}(A, B)$ cu $f : A \rightarrow B$ sau $A \xrightarrow{f} B$
- **Compunere:** pentru orice săgeți $f : A \rightarrow B$ și $g : B \rightarrow C$ există o săgeată $g \circ f : A \rightarrow C$

O categorie

O categorie **C** constă în

- **Obiecte:** notate A, B, C, \dots
- **Săgeți:** pentru orice obiecte A și B , există o mulțime de săgeți $\mathbf{C}(A, B)$
 - notăm $f \in \mathbf{C}(A, B)$ cu $f : A \rightarrow B$ sau $A \xrightarrow{f} B$
- **Compunere:** pentru orice săgeți $f : A \rightarrow B$ și $g : B \rightarrow C$ există o săgeată $g \circ f : A \rightarrow C$

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ & \searrow g \circ f & \downarrow g \\ & & C \end{array}$$

O categorie

O categorie **C** constă în

- **Obiecte:** notate A, B, C, \dots
- **Săgeți:** pentru orice obiecte A și B , există o mulțime de săgeți $\mathbf{C}(A, B)$
 - notăm $f \in \mathbf{C}(A, B)$ cu $f : A \rightarrow B$ sau $A \xrightarrow{f} B$
- **Compunere:** pentru orice săgeți $f : A \rightarrow B$ și $g : B \rightarrow C$ există o săgeată $g \circ f : A \rightarrow C$

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ & \searrow g \circ f & \downarrow g \\ & & C \end{array}$$

- **Identitate:** pentru orice obiect A există o săgeată $\text{id}_A : A \rightarrow A$

O categorie

O categorie **C** constă în

- **Obiecte:** notate A, B, C, \dots
- **Săgeți:** pentru orice obiecte A și B , există o mulțime de săgeți $\mathbf{C}(A, B)$
 - notăm $f \in \mathbf{C}(A, B)$ cu $f : A \rightarrow B$ sau $A \xrightarrow{f} B$
- **Compunere:** pentru orice săgeți $f : A \rightarrow B$ și $g : B \rightarrow C$ există o săgeată $g \circ f : A \rightarrow C$

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ & \searrow g \circ f & \downarrow g \\ & & C \end{array}$$

- **Identitate:** pentru orice obiect A există o săgeată $\text{id}_A : A \rightarrow A$
- **Axiome:** pentru orice săgeți $f : A \rightarrow B$, $g : B \rightarrow C$ și $h : C \rightarrow D$

$$h \circ (g \circ f) = (h \circ g) \circ f \quad f \circ \text{id}_A = f = \text{id}_B \circ f$$

Exemplu - categoria de mulțimi

Categoria **Set** are

- **Obiecte:** mulțimi
- **Săgeți:** funcții
- **Compunere:** compunerea de funcții
- **Identitate:** pentru orice mulțime A , funcția identitate $\text{id}_A : A \rightarrow A$,
 $\text{id}_A(a) = a$
- **Axiome:** ✓

Exemplu - categoria de monoizi

Categoria **Mon** are

- Obiecte: monoizi
- Săgeți: morfisme de monoizi
(aka funcții care nu "strică" operația de monoid)
- Compunerea: compunerea de morfisme de monoizi
- Identitatea: pentru orice obiect \mathbf{M} , $\text{id}_{\mathbf{M}} : M \rightarrow M$, $\text{id}_{\mathbf{M}}(m) = m$
- Axiome: ✓

Exemplu - un monoid ca o categorie

Orice monoid $\mathbf{M} = \langle M, +, e \rangle$ este o categorie cu

- Obiecte: un singur obiect □
- Săgeți: elementele mulțimii M (i.e, $\mathbf{M}(\square, \square) = M$)
- Compunerea: operația de monoid $+$
- Identitatea: identitatea monoidului e
- Axiome:

$$\begin{array}{ll} h \circ (g \circ f) = (h \circ g) \circ f & f \circ id_A = f = id_B \circ f \\ a + (b + c) = (a + b) + c & a + e = a = e + a \end{array}$$

Obiect terminal și obiect inițial

Într-o categorie **C**

- un obiect T se numește **terminal** dacă pentru orice obiect A există o unică săgeată

$$\tau_A : A \rightarrow T$$

Obiect terminal și obiect inițial

Într-o categorie **C**

- un obiect T se numește **terminal** dacă pentru orice obiect A există o unică săgeată

$$\tau_A : A \rightarrow T$$

- un obiect I se numește **inițial** dacă pentru orice obiect A există o unică săgeată

$$\iota_A : I \rightarrow A$$

De ce obiect terminal?

Putem generaliza regulile de mai devreme. De exemplu:

$$\frac{A \quad B}{A \& B} \quad (\&_I)$$

De ce obiect terminal?

Putem generaliza regulile de mai devreme. De exemplu:

$$\frac{A \quad B}{A \& B} (\&_I)$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} (\&_I)$$

Deductie din ipotezele Γ

De ce obiect terminal?

Putem generaliza regulile de mai devreme. De exemplu:

$$\frac{A \quad B}{A \& B} (\&_I)$$

Deducție fără ipoteze
 $\Gamma = \emptyset$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} (\&_I)$$

Deducție din ipotezele Γ

De ce obiect terminal?

Putem generaliza regulile de mai devreme. De exemplu:

$$\frac{A \quad B}{A \& B} (\&_I)$$

Deducre fără ipoteze
 $\Gamma = \emptyset$ (Deducre din adevăr)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} (\&_I)$$

Deducre din ipotezele Γ

De ce obiect terminal?

Putem generaliza regulile de mai devreme. De exemplu:

$$\frac{A \quad B}{A \& B} (\&_I)$$

Deducre fără ipoteze
 $\Gamma = \emptyset$ (Deducre din adevăr)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} (\&_I)$$

Deducre din ipotezele Γ

Fie **C** o categorie cu obiect terminal T .

De ce obiect terminal?

Putem generaliza regulile de mai devreme. De exemplu:

$$\frac{A \quad B}{A \& B} (\&_I)$$

Deducre fără ipoteze
 $\Gamma = \emptyset$ (Deducre din adevăr)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} (\&_I)$$

Deducre din ipotezele Γ

Fie **C** o categorie cu obiect terminal T . Avem următoarele interpretări:

De ce obiect terminal?

Putem generaliza regulile de mai devreme. De exemplu:

$$\frac{A \quad B}{A \& B} (\&_I)$$

Deducre fără ipoteze
 $\Gamma = \emptyset$ (Deducre din adevăr)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} (\&_I)$$

Deducre din ipotezele Γ

Fie **C** o categorie cu obiect terminal T . Avem următoarele interpretări:

- Formulele sunt obiectele din **C**

De ce obiect terminal?

Putem generaliza regulile de mai devreme. De exemplu:

$$\frac{A \quad B}{A \& B} (\&_I)$$

Deducre fără ipoteze
 $\Gamma = \emptyset$ (Deducre din adevăr)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} (\&_I)$$

Deducre din ipotezele Γ

Fie **C** o categorie cu obiect terminal T . Avem următoarele interpretări:

- Formulele sunt obiectele din **C**
- Adevărul este obiectul terminal T

De ce obiect terminal?

Putem generaliza regulile de mai devreme. De exemplu:

$$\frac{A \quad B}{A \& B} (\&_I)$$

Deducre fără ipoteze
 $\Gamma = \emptyset$ (Deducre din adevăr)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} (\&_I)$$

Deducre din ipotezele Γ

Fie **C** o categorie cu obiect terminal T . Avem următoarele interpretări:

- Formulele sunt obiectele din **C**
- Adevărul este obiectul terminal T
- O demonstrație a lui A este o săgeată $f : T \rightarrow A$

De ce obiect terminal?

Putem generaliza regulile de mai devreme. De exemplu:

$$\frac{A \quad B}{A \& B} (\&_I)$$

Deducre fără ipoteze
 $\Gamma = \emptyset$ (Deducre din adevăr)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} (\&_I)$$

Deducre din ipotezele Γ

Fie **C** o categorie cu obiect terminal T . Avem următoarele interpretări:

- Formulele sunt obiectele din **C**
- Adevărul este obiectul terminal T
- O demonstrație a lui A este o săgeată $f : T \rightarrow A$
- O demonstrație a lui A din ipoteza B este o săgeată $f : B \rightarrow A$

Produs

Fie A și B două obiecte în categoria **C**.

Produs

Fie A și B două obiecte în categoria **C**. Spunem că

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$$

este **produsul** lui A și B

Produs

Fie A și B două obiecte în categoria **C**. Spunem că

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$$

este **produsul** lui A și B dacă pentru orice

$$A \xleftarrow{f} C \xrightarrow{g} B$$

Produs

Fie A și B două obiecte în categoria **C**. Spunem că

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$$

este **produsul** lui A și B dacă pentru orice

$$A \xleftarrow{f} C \xrightarrow{g} B$$

există o unică săgeată

$$\langle f, g \rangle : C \longrightarrow A \times B$$

Produs

Fie A și B două obiecte în categoria **C**. Spunem că

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$$

este **produsul** lui A și B dacă pentru orice

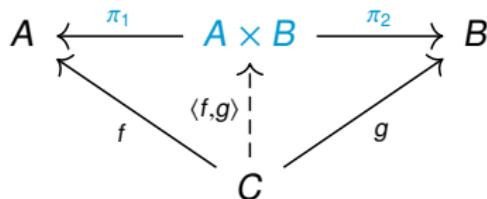
$$A \xleftarrow{f} C \xrightarrow{g} B$$

există o unică săgeată

$$\langle f, g \rangle : C \longrightarrow A \times B$$

astfel încât

$$\pi_1 \circ \langle f, g \rangle = f \quad \pi_2 \circ \langle f, g \rangle = g$$



Să schimbăm iar perspectiva!

Fie **C** o categorie cu obiect terminal și produse.

Fie A, B două obiecte în **C**.

Să schimbăm iar perspectiva!

Fie \mathbf{C} o categorie cu obiect terminal și produse.

Fie A, B două obiecte în \mathbf{C} .

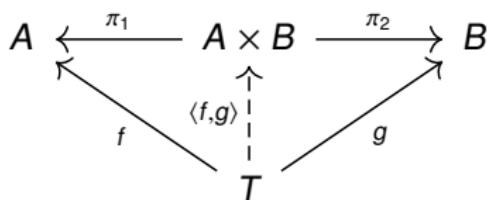
$$\begin{array}{ccccc} A & \xleftarrow{\pi_1} & A \times B & \xrightarrow{\pi_2} & B \\ f \swarrow & \uparrow \langle f, g \rangle & \downarrow & \searrow g & \\ T & & & & \end{array}$$

Să schimbăm iar perspectiva!

Fie \mathbf{C} o categorie cu obiect terminal și produse.

Fie A, B două obiecte în \mathbf{C} .

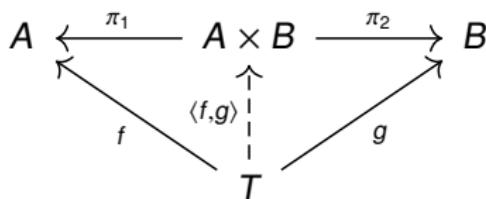
$$\frac{f : T \rightarrow A \quad g : T \rightarrow B}{\langle f, g \rangle : T \rightarrow A \times B}$$



Să schimbăm iar perspectiva!

Fie \mathbf{C} o categorie cu obiect terminal și produse.

Fie A, B două obiecte în \mathbf{C} .



$$\frac{f : T \rightarrow A \quad g : T \rightarrow B}{\langle f, g \rangle : T \rightarrow A \times B}$$

$$\frac{\langle f, g \rangle : T \rightarrow A \times B}{\pi_1 \circ \langle f, g \rangle : T \rightarrow A}$$

Să schimbăm iar perspectiva!

Fie \mathbf{C} o categorie cu obiect terminal și produse.

Fie A, B două obiecte în \mathbf{C} .

$$\begin{array}{ccccc} & & A \times B & & \\ & \swarrow f & \uparrow \langle f,g \rangle & \searrow g & \\ A & \xleftarrow{\pi_1} & & \xrightarrow{\pi_2} & B \end{array}$$

$$\frac{f : T \rightarrow A \quad g : T \rightarrow B}{\langle f, g \rangle : T \rightarrow A \times B}$$

$$\frac{\langle f, g \rangle : T \rightarrow A \times B}{\pi_1 \circ \langle f, g \rangle : T \rightarrow A}$$

$$\frac{\langle f, g \rangle : T \rightarrow A \times B}{\pi_2 \circ \langle f, g \rangle : T \rightarrow B}$$

Să schimbăm iar perspectiva!

Fie \mathbf{C} o categorie cu obiect terminal și produse.

Fie A, B două obiecte în \mathbf{C} .

$$\begin{array}{ccccc} & & A \times B & & \\ & \swarrow f & \uparrow \langle f,g \rangle & \searrow g & \\ A & \xleftarrow{\pi_1} & & \xrightarrow{\pi_2} & B \end{array}$$

$$\frac{f : T \rightarrow A \quad g : T \rightarrow B}{\langle f, g \rangle : T \rightarrow A \times B}$$

$$\frac{\langle f, g \rangle : T \rightarrow A \times B}{\pi_1 \circ \langle f, g \rangle : T \rightarrow A}$$

$$\frac{\langle f, g \rangle : T \rightarrow A \times B}{\pi_2 \circ \langle f, g \rangle : T \rightarrow B}$$

Arată cunoscut?

Ce am văzut până acum

Un λ -calcul cu tipuri

$$\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B} (\times_I)$$

$$\frac{p : A \times B}{fst\ p : A} (\times_{E_1})$$

$$\frac{p : A \times B}{snd\ p : B} (\times_{E_2})$$

Un sist. de deducție naturală

$$\frac{A \quad B}{A \& B} (\&_I)$$

$$\frac{A \& B}{A} (\&_{E_1})$$

$$\frac{A \& B}{B} (\&_{E_2})$$

O categorie*

$$\frac{f : T \rightarrow A \quad g : T \rightarrow B}{\langle f, g \rangle : T \rightarrow A \times B}$$

$$\frac{\langle f, g \rangle : T \rightarrow A \times B}{\pi_1 \circ \langle f, g \rangle : T \rightarrow A}$$

$$\frac{\langle f, g \rangle : T \rightarrow A \times B}{\pi_2 \circ \langle f, g \rangle : T \rightarrow B}$$

* O categorie cu obiect terminal T și produse

Corespondența Curry-Howard-Lambek

Teoria Tipurilor	Logică	Teoria categoriilor
tipuri	formule	obiecte

Corespondența Curry-Howard-Lambek

Teoria Tipurilor	Logică	Teoria categoriilor
tipuri termeni	formule demonstrații	obiecte săgeți

Corespondența Curry-Howard-Lambek

Teoria Tipurilor	Logică	Teoria categoriilor
tipuri termeni	formule demonstrații	obiecte săgeți
tipul unit	true	obiect terminal T

Corespondența Curry-Howard-Lambek

Teoria Tipurilor	Logică	Teoria categoriilor
tipuri	formule	obiecte
termeni	demonstrații	săgeți
<i>inhabitation</i> a tipului A	demonstrație a lui A	săgeată $f : T \rightarrow A$
tipul unit	true	obiect terminal T

Corespondența Curry-Howard-Lambek

Teoria Tipurilor	Logică	Teoria categoriilor
tipuri	formule	obiecte
termeni	demonstrații	săgeți
<i>inhabitation</i> a tipului A	demonstrație a lui A	săgeată $f : T \rightarrow A$
tip produs	conjuncție	produs
tipul unit	true	obiect terminal T

Corespondența Curry-Howard-Lambek

Teoria Tipurilor	Logică	Teoria categoriilor
tipuri	formule	obiecte
termeni	demonstrații	săgeți
<i>inhabitation</i> a tipului A	demonstrație a lui A	săgeată $f : T \rightarrow A$
tip produs	conjuncție	produs
tip sumă	disjuncție	coprodus
tipul unit	true	obiect terminal T

Corespondența Curry-Howard-Lambek

Teoria Tipurilor	Logică	Teoria categoriilor
tipuri	formule	obiecte
termeni	demonstrații	săgeți
<i>inhabitation</i> a tipului A	demonstrație a lui A	săgeată $f : T \rightarrow A$
tip produs	conjuncție	produs
tip sumă	disjuncție	coprodus
tipul void	false	obiect inițial
tipul unit	true	obiect terminal T

Corespondența Curry-Howard-Lambek

Teoria Tipurilor	Logică	Teoria categoriilor
tipuri	formule	obiecte
termeni	demonstrații	săgeți
<i>inhabitation</i> a tipului A	demonstrație a lui A	săgeată $f : T \rightarrow A$
tip funcție	implicație	exponenți
tip produs	conjuncție	produs
tip sumă	disjuncție	coprodus
tipul void	false	obiect inițial
tipul unit	true	obiect terminal T

Referinje

- Roger Antonsen, *TED Talk: Math is the hidden secret to understanding the world*
https://www.ted.com/talks/roger_antonsen_math_is_the_hidden_secret_to_understanding_the_world
- Samson Abramsky, *Categories, Proofs and Processed Lecture III - The Curry-Howard-Lambek Correspondence*
<http://www.math.helsinki.fi/logic/sellc-2010/course/LectureIII.pdf>
- Philip Wadler, *Propositions as Types*
<https://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf>
- Dan Grossman, *Lecture notes on The Curry-Howard Isomorphism*
https://courses.cs.washington.edu/courses/cse505/12au/lec12_6up.pdf

Baftă la examen!