Curs 1

2021-2022

Fundamentele Limbajelor de Programare

Cuprins

Organizare

Instructori

Curs

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

Laborator

Seria 23
- **Ana Iova (urlea) (231)**
- **Horatiu Cheval (232)**
- **Bogdan Macovei (233,234)**

Seria 24
- **Natalia Ozunu (241)**
- **Bogdan Macovei (242,243,244)**

Seria 25
- **Ana Iova (urlea) (251)**
- **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 postate pe Moodle si pe paginile externe

Prezenta

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

Notare

Notare

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

Examen

- In sesiunea
- Durata 2 ore
- Cu materialele ajutatoare
- Mai multe detalii vor fi oferite pana la jumatatea semestrului

Curs/laborator

- Curs
  - Bazele programrii logice
    - Logica clauzelor Horn, Unificare, Rezolutie
  - Semantica limbajelor de programare
    - Semantic operaional, static i axiomatic
    - Inferarea automat a tipurilor
  - Bazele programrii funcionale
    - Lambda Calcul, Codificri Church, corespondenta Curry-Howard
    - Lambda Calcul cu tipuri de date
- Laborator/Seminar:
  - Prolog Cel mai cunoscut limbaj de programare logica
    - Verificator pentru un mini-limbaj imperativ
    - Inferena tipurilor pentru un mini-limbaj funcional
  - Haskell Limbaj pur de programare funcional
    - Interpretoare pentru mini-limbaje
  - 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 programrii funcionale / logice

Principalele paradigme de programare

- Imperativ (<u>cum</u> calculm)
  - Procedural
  - Orientat pe obiecte
- Declarativ (<u>ce</u> calculm)
  - Logic
  - Functional

Fundamentele paradigmelor de programare

Imperativ	Execuia unei Maini Turing
Logic	Rezoluia în logica clauzelor Horn
Funcional	Beta-reducie în Lambda Calcul

Programare declarativ

- Programatorul spune ce vrea s calculeze, dar nu specific concret cum calculeaz.
- Este treaba interpretorului (compilator/implementare) s identifice cum s efectueze calculul respectiv.
- Tipuri de programare declarativ:
  - Programare functional (e.g., Haskell)
  - Programare logic (e.g., Prolog)
  - Limbaje de interogare (e.g., SQL)

Programare funcional

Esen: funcii care relaioneaz intrrile cu ieirile

Caracteristici:
- ☐ funcii de ordin înalt – funcii parametrizate de funcii
- ☐ grad înalt de abstractizare (e.g., functori, monade)
- ☐ grad înalt de reutilizarea codului — polimorfism

Fundamente:
- ☐ Teoria funciilor recursive
- ☐ Lambda-calcul ca model de computabilitate (echivalent cu maina Turing)

Inspiraie:
- ☐ Inferena tipurilor pentru templates/generics in POO
- ☐ Model pentru programarea distribuit/bazat pe evenimente (callbacks)

Programare logic

- Programarea logic este o paradigm de programare bazat pe logic formal.
- Unul din sloganurile programrii logice:

  Program = Logic + Control (R. Kowalski)

- Programarea logic poate fi privit ca o deductie controlat.
- Un program scris intr-un limbaj de programare logic este

  o list de formule intr-o logic

  ce exprim fapte i reguli despre o problem.
- Exemple de limbaje de programare logic:
  - Prolog
  - Answer set programming (ASP)
  - Datalog

Semantica Limbajelor de Programare

Ce definete un limbaj de programare?

Sintaxa Simboluri de operaie, cuvinte cheie, descriere (formal) a programelor/expresiilor bine formate

Practica Un limbaj e definit de modul cum poate fi folosit

- Manual de utilizare i exemple de bune practici
- Implementare (compilator/interpretor)
- Instrumente ajuttoare (analizor de sintax, verificator de tipuri, depanator)

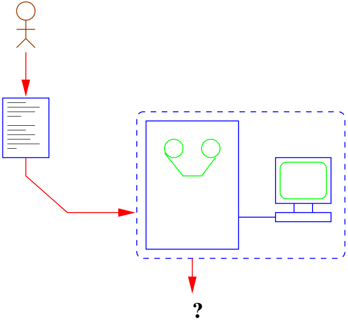
Semantica? Ce înseamn / care e comportamentul unei instruciuni?

- De cele mai multe ori se d din umeri i se spune c Practica e suficient
- Limbajele mai utilizate sunt standardizate

La ce folosete semantica

- S înelegem un limbaj în profunzime
  - Ca programator: pe ce m pot baza când programez în limbajul dat
  - Ca implementator al limbajului: ce garanii trebuie s ofer
- Ca instrument în proiectarea unui nou limbaj / a unei extensii
  - Înelegerea componentelor i a relaiilor dintre ele
  - Exprimarea (i motivarea) deciziilor de proiectare
  - Demonstrarea unor proprieti generice ale limbajului
    E.g., execuia nu se va bloca pentru programe care trec de analiza tipurilor
- Ca baz pentru demonstrarea corectitudinii programelor.

Problema corectitudinii programelor



- Pentru anumite metode de programare (e.g., imperativ, orientat pe obiecte), nu este uor s stabilim c un program este corect sau s înelegem ce înseamn c este corect (e.g, în raport cu ce?!).
- Corectitudinea programelor devine o problem din ce în ce mai important, nu doar pentru aplicaii "safety-critical".
- Avem nevoie de metode ce asigur "calitate", capabile s ofere "garanii".

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?
- Un formalism adecvat trebuie:
  - s permit descrierea problemelor (specificaii), i
  - s raioneze despre implementarea lor (corectitudinea programelor).

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 întoars e 4
- GCC3, ICC, Clang: valoarea întoars e 3

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 i 2.

Tipuri de semantic

Semantica d "îneles" unui program.

- Operaional:
  - Înelesul programului este definit în funcie de paii (transformri dintr-o stare în alta) care apar în timpul execuiei.
- Denotaional:
  - Înelesul programului este definit abstract ca element dintr-o structur matematic adecvat.
- Axiomatic:
  - Înelesul programului este definit indirect în funcie de axiomele i regulile pe care le verific.
- Static / a tipurilor
  - Reguli de bun-formare pentru programe
  - Ofer garanii privind execuia (e.g., nu se blocheaz)

Programare logic & Prolog

Programare logic - în mod idealist

- Un "program logic" este o colecie de proprieti presupuse (sub form de formule logice) despre lume (sau mai degrab despre lumea programului).
- Programatorul furnizeaz i o proprietate (o formula logic) care poate s fie sau nu adevrat în lumea respectiv (întrebare, query).
- Sistemul determin dac proprietatea aflat sub semnul întrebrii este o consecin a proprietilor presupuse în program.
- Programatorul nu specific metoda prin care sistemul verific dac întrebarea este sau nu consecin a programului.

Exemplu de program logic

$$\begin{array}{rcl}
\texttt{oslo} & \to & \texttt{windy} \\
\texttt{oslo} & \to & \texttt{norway} \\
\texttt{norway} & \to & \texttt{cold} \\
\texttt{cold} \land \texttt{windy} & \to & \texttt{winterIsComing} \\
 & & \texttt{oslo}
\end{array}$$

Exemplu de întrebare

Este adevrat `winterIsComing`?

Prolog

- bazat pe logica clauzelor Horn
- semantica operaional este bazat pe rezoluie
- este Turing complet

Limbajul Prolog este folosit pentru programarea sistemului IBM Watson!

Putei 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

Sintax: constante, variabile, termeni compui

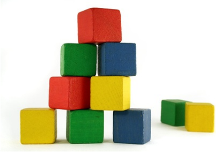
- Atomi: `brian, 'Brian Griffin', brian_griffin`

- Numere: `23, 23.03,-1`

  Atomii i numerele sunt constante.

- Variabile: `X, Griffin, _family`

- Termeni compui: `father(peter, stewie_griffin),`
  `and(son(stewie,peter), daughter(meg,peter))`
  - forma general: atom(termen,. . ., termen)
  - atom-ul care denumete termenul se numete functor
  - numrul de argumente se numete aritate



Un mic exerciiu sintactic

Care din urmtoarele iruri de caractere sunt constante i care sunt variabile în Prolog?

- vINCENT – constant
- Footmassage – variabil
- variable23 – constant
- Variable2000 – variabil
- big_kahuna_burger – constant
- 'big kahuna burger' – constant
- big kahuna burger – nici una, nici alta
- 'Jules' – constant
- _Jules – variabil
- '_Jules' – constant

Program în Prolog = baz de cunotine

Exemplu

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 baz de cunotine (Knowledge Base).

Program în Prolog = mulime de predicate

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

Exemplu

```
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

- Un program în Prolog este format din reguli de forma

`Head :- Body.`

- Head este un predicat, iar Body este o secven de predicate separate prin virgul.

- Regulile fr `Body` se numesc fapte.

Exemplu

- Exemplu de regul: `griffin(X) :- father(Y,X), griffin(Y).`
- Exemplu de fapt: `father(peter,meg).`

Interpretarea din punctul de vedere al logicii

- operatorul **:-** este implicaia logic $\leftarrow$

Exemplu

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

dac `griffin(X)` este adevrat, atunci `comedy(X)` este adevrat.

- virgula , este conjuncia $\wedge$

Exemplu

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

dac `father(Y,X)` i `griffin(Y)` sunt adevrate,
atunci `griffin(X)` este adevrat.

Interpretarea din punctul de vedere al logicii

- mai multe reguli cu acelai `Head` definesc acelai predicat, între defiii fiind un sau logic.

Exemplu

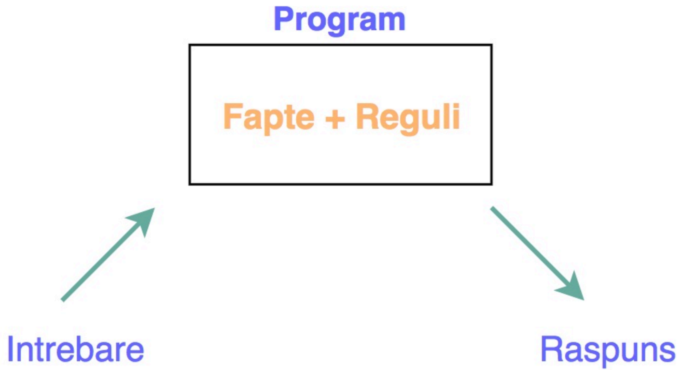
```
comedy(X) :- family_guy(X).
comedy(X) :- south_park(X).
comedy(X) :- disenchantment(X).
```

dac
`family_guy(X)` este adevrat sau `south_park(X)` este adevrat sau `disenchantment(X)` este adevrat,
atunci
`comedy(X)` este adevrat.

Un program în Prolog

Cum folosim un program în Prolog?

Întrebri în Prolog



Întrebri i inte în Prolog

- Prolog poate rspunde la întrebri legate de consecinele relaiilor descrise într-un program în Prolog.
- Întrebrile sunt de forma:

$$\texttt{?- predicat}_1\texttt{(...),...,predicat}_n\texttt{(...).}$$

- Prolog verific dac întrebarea este o consecin a relaiilor definite în program.
- Dac este cazul, Prolog caut valori pentru variabilele care apar în întrebare astfel încât întrebarea s fie o consecin a relaiilor din program.
- Un predicat care este analizat pentru a se rspunde la o întrebare se numete int (goal).

Întrebri în Prolog

Prolog poate da 2 tipuri de rspunsuri:

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

Exemplu

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

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

Pe sptmâna viitoare!

Curs 2

Cuprins

Programare logică & Prolog

Program în Prolog = mulțime de predicate

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

Example

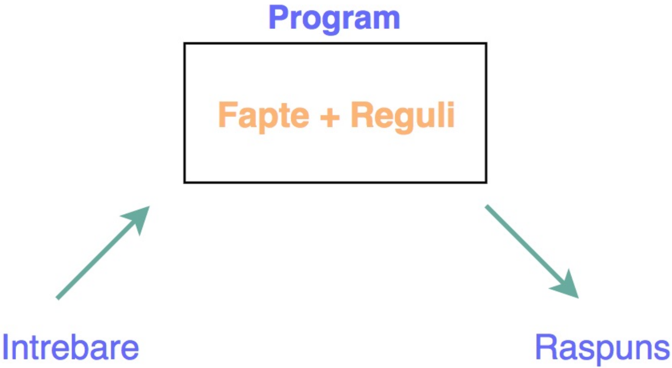
```
father(peter,meg).
father(peter,stewie).

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

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.

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 raspuns, 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.
```

Pentru a răspunde la întrebare se caută o potrivire (unificator) între scopul `foo(X)` și baza de cunoștințe. Raspunsul 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 raspuns, 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
```

Dacaă nu se poate face potrivirea, răspunsul este false.

Cum găsește Prolog răspunsul

Pentru a găsi un raspuns, 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 raspuns, 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 raspuns, 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 raspuns, Prolog redenumește variabilele.

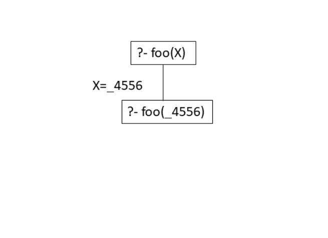
Example

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 raspuns, Prolog încearcă regulile în ordinea apariției lor.

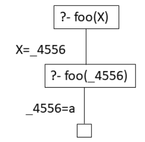
Example

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 raspuns, Prolog încearcă clauzele în ordinea apariției lor.

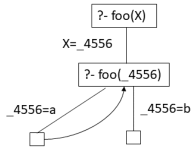
Example

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 țintei cu o nouă valoare.

Cum găsește Prolog răspunsul

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

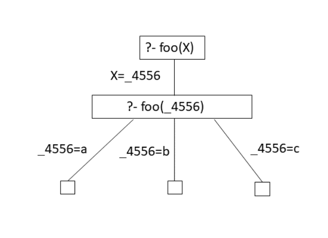
Example

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

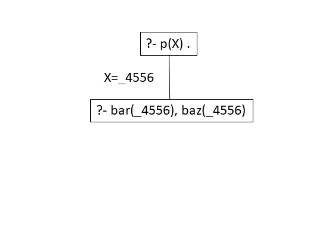
Example

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.

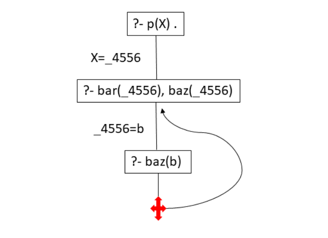
Example

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-țintă eșuează.

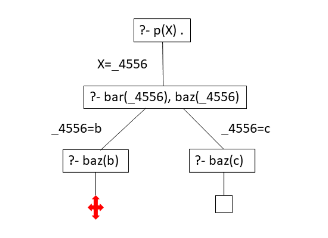
Example

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
```

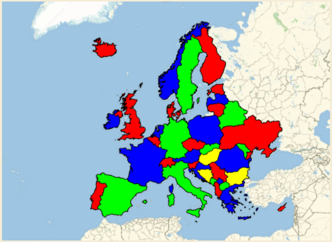
Vă explicați ce s-a întâmplat? Desenați arborele de căutare!

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

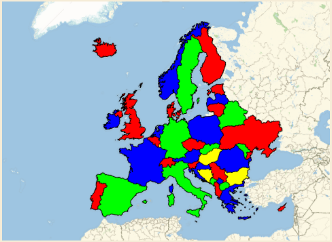
Un program mai complicat

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

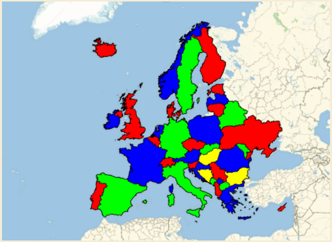
Să se coloreze o hartă dată cu un număr minim de culori astfel încât oricare două țări vecine să fie colorate diferit.

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(RO,SE,MD,UA,BG,HU) :- vecin(RO,SE), vecin(RO,UA),
                            vecin(RO,MD), vecin(RO,BG),
                            vecin(RO,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(RO,SE,MD,UA,BG,HU) :- vecin(RO,SE), vecin(RO,UA),
                            vecin(RO,MD), vecin(RO,BG),
                            vecin(RO,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(RO,SE,MD,UA,BG,HU) :- vecin(RO,SE), vecin(RO,UA),
                            vecin(RO,MD), vecin(RO,BG),
                            vecin(RO,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(RO,SE,MD,UA,BG,HU) :- vecin(RO,SE), vecin(RO,UA),
                            vecin(RO,MD), vecin(RO,BG),
                            vecin(RO,HU), vecin(UA,MD),
                            vecin(BG,SE), vecin(SE,HU).
vecin(X,Y) :- culoare(X),
              culoare(Y),
              X \== Y.

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

Problema colorării hărților

Ce răspuns primim?

Example

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

harta(RO,SE,MD,UA,BG,HU) :- vecin(RO,SE), vecin(RO,UA),
                            vecin(RO,MD), vecin(RO,BG),
                            vecin(RO,HU), vecin(UA,MD),
                            vecin(BG,SE), vecin(SE,HU).
vecin(X,Y) :- culoare(X),
              culoare(Y),
              X \== Y.

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

Problema colorării hărților

Example

```
culoare(albastru).
culoare(rosu).
culoare(verde).
culoare(galben).
harta(RO,SE,MD,UA,BG,HU) :-   vecin(RO,SE), vecin(RO,UA),
                              vecin(RO,MD), vecin(RO,BG),
                              vecin(RO,HU), vecin(UA,MD),
                              vecin(BG,SE), vecin(SE,HU).
vecin(X,Y) :- culoare(X),
              culoare(Y),
              X \== Y.
```

```
?- harta(RO,SE,MD,UA,BG,HU).
RO = albastru,
SE = UA, UA = rosu,
MD = BG, BG = HU, HU = verde
```

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

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

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

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

Example

```
?- X = Y.
X = Y.

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

?- 2 = 1 + 1
false
```

```
?- X == Y .
false

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

?- 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.

Negarea unui predicat: `\+ pred(X)`

Example

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

?- animal(cat).
false
?- \+ 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 țintă, Prolog trebuie să construiască o "demonstrație" pentru a arată că mulțimea de fapte și reguli din program implică acea țintă.
- Astfel, un răspuns `false` nu înseamnă neapărat că ținta nu este adevărată, ci doar că Prolog nu a reușit să găsească o demonstrație.

Operatorul \+

- Negarea unei ținte se poate defini astfel:

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

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

Operatorul \+

- Negarea unei ținte se poate defini astfel:

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

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

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

Negația 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).
```

Negația 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 -> /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 -> Then` este echivalent cu `If -> Then ; fail`.

Liste și recursie

Listă [t1,...,tn]

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

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

- O listă poate conține 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,...,tn]==[t1 |[t2,...,tn]

- 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 șabloane mai complicate:

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

Exercițiu

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

Liste

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).
```

Liste

Exercițiu

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.

Liste

Exercițiu

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

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).
```

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

Exerciții

- 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).
```

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

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.
```

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 acumulatori

- Varianta inițială:

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

- Varianta cu acumulator

```
rev(L,R) :- revac(L,[],R).
% la momentul inițial nu am acumulat nimic.
```

Recursie cu acumulatori

- Varianta inițială:

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

- Varianta cu acumulator

```
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 acumulatori

- Varianta inițială:

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

- Varianta cu acumulator

```
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.

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(0,[]).
biglist(N,[N|T]) :- N >= 1, M is N-1,biglist(M,T),M=M.

biglist_tr(0,[]).
biglist_tr(N,[N|T]) :- N >= 1, M is N-1,biglist_tr(M,T).
```

Recursie la coadă

- Predicat fără recursie la coadă:

```
biglist(0,[]).
biglist(N,[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(0,[]).
biglist(N,[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(0,[]).
biglist(N,[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(0,[]).
biglist_tr(N,[N|T]) :- N >= 1, M is N-1,biglist_tr(M,T).
```

Recursie la coadă

- Predicat fără recursie la coadă:

```
biglist(0,[]).
biglist(N,[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(0,[]).
biglist_tr(N,[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|...]
```

Tipuri de date compuse

Termeni compuși `f(t1,..., tn)`

- 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!

Arbori binari în Prolog

- Cum arată un arbore?

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

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

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

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

```
def(arb, tree(a, tree(b,
                      tree(d,void,void),
                      void),
             tree(c, void,
                      tree(e,void,void)))).
```

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

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

```
def(arb, tree(a, tree(b,
                        tree(d,void,void),
                        void),
                  tree(c, void,
                        tree(e,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.

Arbori binari în Prolog

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

Arbori 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).
```

Arbori 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).
```

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 */
```

Arbori 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).
```

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).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care verifică că un element aparține unui arbore.

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care verifică că 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).
```

Arbori binari în Prolog

Exercițiu

Scrieți 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).
```

Arbori binari în Prolog

Exercițiu

Scrieți 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,[]).
```

Arbori binari în Prolog

Exercițiu

Scrieți 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).
```

Arbori binari în Prolog

Exercițiu

Scrieți 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(a, tree(b, tree(d, void, void), void), tree(c,
void, tree(e, void, void))),
P = [a, b, d, c, e]
```

Quiz time!

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

Exemplu: reprezentarea unei GIC

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. Chomsy, Syntactic structure, Mouton Publishers, First printing 1957 - Fourteenth printing 1985 [Chapter 4 (Phrase Structure)]

  (i) $Sentence \rightarrow NP + VP$
  (ii) $NP \rightarrow T + N$
  (iii) $VP \rightarrow Verb + NP$
  (iv) $T \rightarrow the$
  (q) $N \rightarrow man, ball$, etc.
  (vi) $V \rightarrow hit, took$, etc.

Gramatică independentă de context

- Definim structura propozițiilor folosind o gramatică independentă de context:

S	$\rightarrow$	NP VP
NP	$\rightarrow$	Det N
VP	$\rightarrow$	V
VP	$\rightarrow$	V NP
Det	$\rightarrow$	*the*
Det	$\rightarrow$	*a*
N	$\rightarrow$	*boy*
N	$\rightarrow$	*girl*
V	$\rightarrow$	*loves*
V	$\rightarrow$	*hates*

- 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

			Det	→	*the*
S	→	NP VP	Det	→	*a*
NP	→	Det N	N	→	*boy*
VP	→	V	N	→	*girl*
VP	→	V NP	V	→	*loves*
			V	→	*hates*

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]`

- Fiecărui 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 → 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

			Det	$\rightarrow$	*the*
S	$\rightarrow$	NP VP	Det	$\rightarrow$	*a*
NP	$\rightarrow$	Det N	N	$\rightarrow$	*boy*
VP	$\rightarrow$	V	N	$\rightarrow$	*girl*
VP	$\rightarrow$	V NP	V	$\rightarrow$	*loves*
			V	$\rightarrow$	*hates*

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.

Pe săptămâna viitoare!

Curs 3

Cuprins

Termeni, substituții, unificatori

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 $\mathcal{V}$:

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

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
- $x, y, z, \ldots$ pentru variabile
- $a, b, c, \ldots$ pentru constante
- $f, g, h, \ldots$ pentru simboluri de functii arbitrare
- $s, t, u, \ldots$ pentru termeni
- $var(t)$ multimea variabilelor care apa in t
- ecuatii $s \doteq t$ pentru o pereche de termeni
- $Trm_{\mathcal{F},\mathcal{V}}$ 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 constanta
- $var(f(x, g(x, a), y)) = \{x, y\}$

Substituții

Definiție

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

$$\sigma : \mathcal{V} \to Trm_{\mathcal{F},\mathcal{V}}$$

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 substitutii σ unui termen t:

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

Substituții

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

Aplicarea unei substitutii σ unui termen t:

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

Exemplu

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

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

Algoritmul de unificare

- Pentru o mulțime finită de termeni $\{t_1, \ldots, t_n\}$, $n \geq 2$, 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 $\{t_1, \ldots, t_n\}$, $n \geq 2$, 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 $\{t_1, \ldots, t_n\}$, $n \geq 2$, 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: $R = \{t_1 \doteq t_2, \ldots, t_{n-1} \doteq t_n\}$

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 R este eliminată.

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 $f(t_1, \ldots, t_n) \doteq f(t'_1, \ldots, t'_n)$ din R este înlocuită cu ecuațiile $t_1 \doteq t'_1, \ldots, t_n \doteq t'_n$.

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 $f(t_1, \ldots, t_n) \doteq f(t'_1, \ldots, t'_n)$ din R este înlocuită cu ecuațiile $t_1 \doteq t'_1, \ldots, t_n \doteq t'_n$.
- 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
$$f(t_1, \ldots, t_n) \doteq g(t'_1, \ldots, t'_k) \text{ cu } f \neq g.$$
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	$\emptyset$	$t_1 \doteq t'_1, \ldots, t_n \doteq t'_n$
SCOATE	S	$R', \; t \doteq t$
	S	R'
DESCOMPUNE	S	$R', \; f(t_1, \ldots, t_n) \doteq f(t'_1, \ldots, t'_n)$
	S	$R', \; t_1 \doteq t'_1, \ldots t_n \doteq t'_n$
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

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

Exemplu

Exemplu

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

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

Exemplu

Exemplu

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

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

Exemplu

Exemplu

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

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

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

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

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

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

- Ecuațiile $\{g(y) \doteq x,\ f(x, h(x), y) \doteq f(g(z), w, z)\}$ 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 $\{g(y) \doteq x,\ f(x, h(y), y) \doteq f(g(z), b, z)\}$ 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
$x \doteq g(y)$	$g(y) \doteq g(z),\ h(y) \doteq b,\ y \doteq z$	- EȘEC -

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
$x \doteq g(y)$	$g(y) \doteq g(z),\ h(y) \doteq b,\ y \doteq z$	- EȘEC -

- h și b sunt simboluri de operații diferite!
- Nu există unificator pentru acesti 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
$x \doteq g(y)$	$g(y) \doteq y,\ h(g(y)) \doteq w,\ y \doteq z$	- EȘEC -

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
$x \doteq g(y)$	$g(y) \doteq y,\ h(g(y)) \doteq w,\ y \doteq z$	- EȘEC -

- 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 sar 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 sar 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

Corectitudinea algoritmului (optional)

Corectitudinea algoritmului

Lema 1

Mulțimea unificatorilor pentru ecuațiile din $R \cup S$ nu se modifică prin aplicarea celor trei reguli ale algoritmului de unificare.

Corectitudinea algoritmului

Lema 1

Mulțimea unificatorilor pentru ecuațiile din $R \cup S$ nu se modifică prin aplicarea celor trei reguli ale algoritmului de unificare.

Demonstrație

Analizăm fiecare regulă:

- SCOATE: evident

Corectitudinea algoritmului

Lema 1

Mulțimea unificatorilor pentru ecuațiile din $R \cup S$ nu se modifică prin aplicarea celor trei reguli ale algoritmului de unificare.

Demonstrație

Analizăm fiecare regulă:

- SCOATE: evident
- DESCOMPUNE: Trebuie să arătăm că

$$\begin{array}{ccc} \nu \text{ unificator pt.} & \Leftrightarrow & \nu \text{ unificator pt.} \\ f(t_1, \ldots, t_n) \doteq f(t'_1, \ldots, t'_n) & & t_i \doteq t'_i, \text{ or. } i = 1, \ldots, n. \end{array}$$

Corectitudinea algoritmului

Lema 1

Mulțimea unificatorilor pentru ecuațiile din $R \cup S$ nu se modifică prin aplicarea celor trei reguli ale algoritmului de unificare.

Demonstrație

Analizăm fiecare regulă:

- SCOATE: evident
- DESCOMPUNE: Trebuie să arătăm că

$$\begin{array}{ccc} \nu \text{ unificator pt.} & \Leftrightarrow & \nu \text{ unificator pt.} \\ f(t_1, \ldots, t_n) \doteq f(t'_1, \ldots, t'_n) & & t_i \doteq t'_i, \text{ or. } i = 1, \ldots, n. \end{array}$$

$$\begin{aligned} & \nu \text{ unif. pt. } f(t_1, \ldots, t_n) \doteq f(t'_1, \ldots, t'_n) \\ & \Leftrightarrow \nu(f(t_1, \ldots, t_n)) = \nu(f(t'_1, \ldots, t'_n)) \\ & \Leftrightarrow f(\nu(t_1), \ldots, \nu(t_n)) = f(\nu(t'_1), \ldots, \nu(t'_n)) \\ & \Leftrightarrow \nu(t_i) = \nu(t'_i), \text{or. } i = 1, \ldots, n \\ & \Leftrightarrow \nu \text{ unificator pt. } t_i \doteq t'_i, \text{or. } i = 1, \ldots, n \end{aligned}$$

Demonstrație (cont.)

- REZOLVĂ:
  - Se observă că orice unificator ν pentru ecuațiile din $R \cup S$, atât înainte cât și după aplicarea regulii REZOLVĂ, trebuie să satisfacă:
  $$\nu(x) = \nu(t).$$
  - Dacă μ este unificator pentru $x \doteq t$ observăm că:
  $$(x \leftarrow t); \mu = \mu$$
  unde $(x \leftarrow t)(x) = t$ și $(x \leftarrow t)(y) = y$ pentru orice $y \neq x \in V$.
    - $((x \leftarrow t); \mu)(x) = \mu(t) = \mu(x)$
    - $((x \leftarrow t); \mu)(y) = \mu(y)$, pentru orice $y \neq x$
  - Deci,

    μ este un unificator pentru ecuațiile din $R \cup S$ înainte de REZOLVĂ

    $\Leftrightarrow$

    μ este un unificator pentru ecuațiile din $R \cup S$ după REZOLVĂ $\square$

Corectitudinea algoritmului

- Pres. că algoritmul de unificare se termină cu $R = \emptyset$.
- Fie $x_i \doteq t_i$, $i = 1, \ldots, k$, ecuațiile din S.
- Variabilele care apar în partea stângă a ecuațiilor din S sunt distincte două câte două și nu mai apar în termenii $t1$, ..., t_k.
- Definim substituția:

$$\nu(x_i) = t_i \text{ pentru orice } i = 1, \ldots, k.$$

- Observăm că $\nu(t_i) = t_i = \nu(x_i)$ oricare $i = 1, \ldots, k$, deci ν este un unificator pentru $R \cup S$.

Corectitudinea algoritmului

Lema 2

ν definit mai sus cf. algoritmului de unificare este cgu pentru $R \cup S$.

Corectitudinea algoritmului

Lema 2

ν definit mai sus cf. algoritmului de unificare este cgu pentru $R \cup S$.

Demonstrație

La ultimul pas $R = \emptyset$ și $\nu(x_i) = t_i$ oricare $i = 1, \ldots, k$

- Fie μ un alt unificator pentru S. Avem
  - $\mu(\nu(x_i)) = \mu(t_i) = \mu(x_i)$, or. $i = 1, \ldots, k$,
  - $\mu(\nu(y)) = \mu(y)$, or. $y \neq x$.

  Deci $\nu; \mu = \mu$. În concluzie, ν este cgu deoarece oricare alt

  unificator se poate scrie ca o compunere a lui ν cu o substituție. $\square$

- Din Lema 1 rezultă că ν este unificator pentru problema inițială $\{u_1 \doteq u_2, \ldots, u_{n-1} \doteq u_n\}$, deci

$$\nu(u_1) = \cdots = \nu(u_n).$$

Complexitatea algoritmului

- Problema de unificare

  $R = \{x_1 \doteq f(x_0, x_0), x_2 \doteq f(x_1, x_1), \ldots, x_n \doteq f(x_{n-1}, x_{n-1})\}$

  are cgu $S = \{x_1 \leftarrow f(x_0, x_0), x_2 \leftarrow f(f(x_0, x_0), f(x_0, x_0)), \ldots\}$.

Complexitatea algoritmului

- Problema de unificare

  $R = \{x_1 \doteq f(x_0, x_0), x_2 \doteq f(x_1, x_1), \ldots, x_n \doteq f(x_{n-1}, x_{n-1})\}$

  are cgu $S = \{x_1 \leftarrow f(x_0, x_0), x_2 \leftarrow f(f(x_0, x_0), f(x_0, x_0)), \ldots\}$.

- La pasul Elimină, pentru a verifica că o variabilă x_i nu apare în membrul drept al ecuației (occur check) facem 2^i comparații.

Complexitatea algoritmului

- Problema de unificare

  $R = \{x_1 \doteq f(x_0, x_0), x_2 \doteq f(x_1, x_1), \ldots, x_n \doteq f(x_{n-1}, x_{n-1})\}$

  are cgu $S = \{x_1 \leftarrow f(x_0, x_0), x_2 \leftarrow f(f(x_0, x_0), f(x_0, x_0)), \ldots\}$.

- La pasul Elimină, pentru a verifica că o variabilă x_i nu apare în membrul drept al ecuației (occur check) facem 2^i comparații.

- Algoritmul de unificare prezentat anterior este exponențial. Complexitatea poate fi îmbunătățită printr-o reprezentare eficientă a termenilor.

  K. Knight, Unification: A Multidisciplinary Survey, ACM Computing Surveys, Vol. 21, No. 1, 1989.

Pe săptămâna viitoare!

Curs 4

Cuprins

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 acumulatori

- Varianta inițială:

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

- Varianta cu acumulator

```
rev(L,R) :- revac(L,[],R).
% la momentul inițial nu am acumulat nimic.
```

Recursie cu acumulatori

- Varianta inițială:

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

- Varianta cu acumulator

```
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 acumulatori

- Varianta inițială:

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

- Varianta cu acumulator

```
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(t1,..., tn)`

- **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 <u>functori</u>
  - `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!

Arbori binari în Prolog

- Cum arată un arbore?

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

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

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

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

```
def(arb, tree(a, tree(b,
                      tree(d,void,void),
                      void),
                 tree(c, void,
                      tree(e,void,void)))).
```

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

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

```
def(arb, tree(a, tree(b,
                         tree(d,void,void),
                         void),
                 tree(c, void,
                         tree(e,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.

Arbori binari în Prolog

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

Arbori 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).
```

Arbori 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).
```

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 */
```

Arbori 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).
```

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).
```

Arbori binari în Prolog

Exercițiu

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

Arbori binari în Prolog

Exercițiu

Scrieți 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).
```

Arbori binari în Prolog

Exercițiu

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

Arbori binari în Prolog

Exercițiu

Scrieți 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).
```

Arbori binari în Prolog

Exercițiu

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

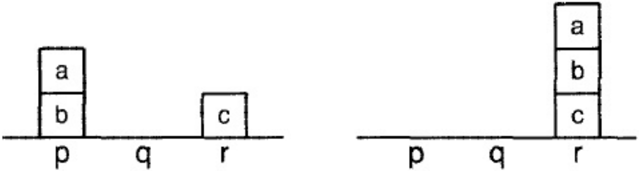
```
preorder(tree(X,L,R),Xs) :- preorder(L,Ls),
                            preorder(R,Rs),
                            append([X|Ls],Rs,Xs).
preorder(void,[]).
```

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

```
?- test(T,P).
T = tree(a, tree(b, tree(d, void, void), void), tree(c,
void, tree(e, void, void))),
P = [a, b, d, c, e]
```

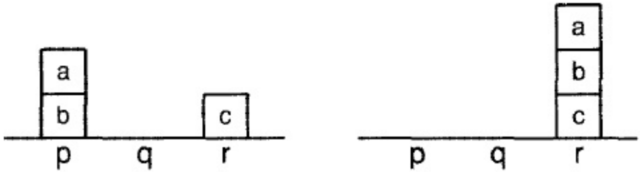
Planning în Prolog

Problemă: Lumea blocurilor



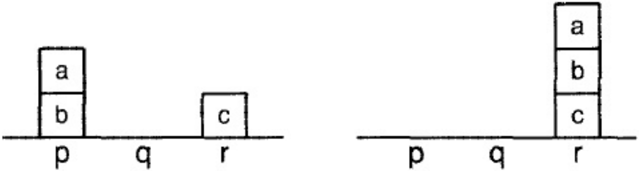
- Lumea blocurilor este formată din:
  - trei blocuri: a,b, c
  - trei poziții: p,q, r
  - un bloc poate sta peste un alt bloc sau pe o poziție

Problemă: Lumea blocurilor



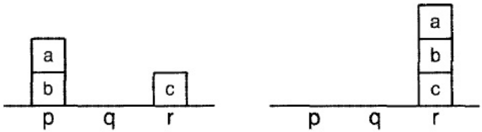
- Lumea blocurilor este formată din:
  - trei blocuri: a,b, c
  - trei poziții: p,q, r
  - un bloc poate sta peste un alt bloc sau pe o poziție
- Un bloc poate fi mutat pe o poziție liberă sau pe un alt bloc.

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 șir de mutări astfel încât dintr-o stare inițială să se ajungă într-o stare finală

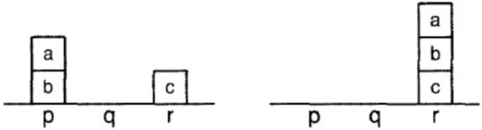
Lumea blocurilor



- Reprezentarea blocurilor, pozițiilor și a stărilor:

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

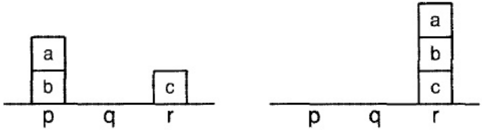
Lumea blocurilor



- 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

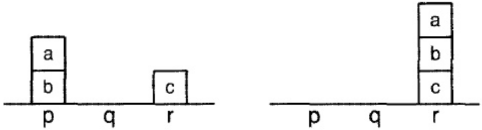


- Reprezentarea blocurilor, pozițiilor și a stărilor:

```
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 șir 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 șir 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

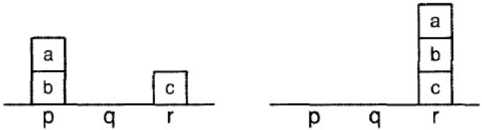
- Predicatul `update(Action,State,State1)` are următoarea semnificație: făcând mutarea `Action` în starea `State` se ajunge în starea `State1`.

```
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`.

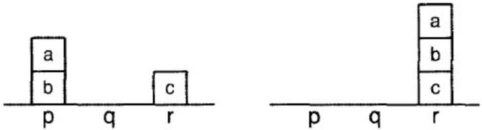
```
substitute(X,Y,[X|Xs],[Y|Xs]).
substitute(X,Y,[X1|Xs],[X1|Ys]) :- X \== X1,
                                   substitute(X,Y,Xs,Ys).
```

Lumea blocurilor



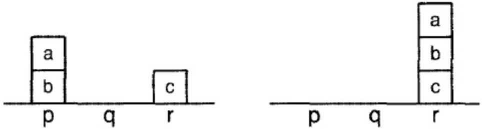

```
block(a). block(b). block(c).
place(p). place(q). place(r).

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



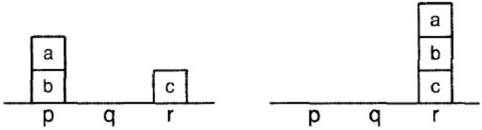

```
test_plan(Plan) :- initial_state(I), final_state(F),
                                       valid_plan(I,F,Plan).
?- test(Plan).
Plan = [to_block(a, c), to_block(b, a), to_place(b, q),
        to_block(a, b), to_block(c, a), to_place(c, p),
        to_block(a, c), to_block(b, a), to_place(b, r),
        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 șir 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. Chomsy, Syntactic structure, Mouton Publishers, First printing 1957 - Fourteenth printing 1985 [Chapter 4 (Phrase Structure)]

  (i) $Sentence \rightarrow NP + VP$
  (ii) $NP \rightarrow T + N$
  (iii) $VP \rightarrow Verb + NP$
  (iv) $T \rightarrow the$
  (q) $N \rightarrow$ *man*, *ball*, etc.
  (vi) $V \rightarrow$ *hit*, *took*, etc.

Gramatică independentă de context

- Definim structura propozițiilor folosind o gramatică independentă de context:

S	$\rightarrow$	NP VP
NP	$\rightarrow$	Det N
VP	$\rightarrow$	V
VP	$\rightarrow$	V NP
Det	$\rightarrow$	*the*
Det	$\rightarrow$	*a*
N	$\rightarrow$	*boy*
N	$\rightarrow$	*girl*
V	$\rightarrow$	*loves*
V	$\rightarrow$	*hates*

- 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

			Det	→	*the*
S	→	NP VP	Det	→	*a*
NP	→	Det N	N	→	*boy*
VP	→	V	N	→	*girl*
VP	→	V NP	V	→	*loves*
			V	→	*hates*

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]
  ```

- Fiecărui 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

			Det	$\rightarrow$	*the*
S	$\rightarrow$	NP VP	Det	$\rightarrow$	*a*
NP	$\rightarrow$	Det N	N	$\rightarrow$	*boy*
VP	$\rightarrow$	V	N	$\rightarrow$	*girl*
VP	$\rightarrow$	V NP	V	$\rightarrow$	*loves*
			V	$\rightarrow$	*hates*

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, așa 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, așa 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 `[t1,...,tn]` va fi reprezentată printr-o pereche

  `([t1,...,tn|T], T)`

  Această pereche poate fi notată `[t1,...,tn|T]- T`, dar notația nu este importantă.

Liste ca diferențe

- Ideea: lista `[t1,...,tn]` va fi reprezentată printr-o pereche

  `([t1,...,tn|T], T)`

  Această pereche poate fi notată `[t1,...,tn|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,...,tn|T], T)

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

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

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

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

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

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

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

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

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

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

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

- Dacă `[t1,..., tn]` este diferența `(X1,T1)`, iar `[q1,..., qk]` este diferența `(X2,T2)` observăm că diferența `(R,T)` trebuie să fie `[t1,...,tn,q1..., qk]`.
- Obținem `R=[t1,...,tn,q1..., qk|T]`, deci `(X1,T1) = (R, P)` și `(X2,T2) = (P,T)` unde `P =[q1,...,qk|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(0,[]).
biglist(N,[N|T]) :- N >= 1, M is N-1,biglist(M,T),M=M.

biglist_tr(0,[]).
biglist_tr(N,[N|T]) :- N >= 1, M is N-1,biglist_tr(M,T).
```

Recursie la coadă

- Predicat fără recursie la coadă:

```
biglist(0,[]).
biglist(N,[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(0,[]).
biglist(N,[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(0,[]).
biglist(N,[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(0,[]).
biglist_tr(N,[N|T]) :- N >= 1, M is N-1,biglist_tr(M,T).
```

Recursie la coadă

- Predicat fără recursie la coadă:

```
biglist(0,[]).
biglist(N,[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(0,[]).
biglist_tr(N,[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

Logica propozițională PL

Logica propozițională PL

- O propoziție este un enunț care poate fi adevărat (1) sau fals (0).
- Propozițiile sunt notate simbolic (φ, ψ, χ, $\cdots$) 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).
- Propozițiile sunt notate simbolic (φ, ψ, χ, $\cdots$) și sunt combinate cu ajutorul conectorilor logici ($\neg$, $\rightarrow$, $\vee$, $\wedge$, $\leftrightarrow$).

Example

Fie φ propoziția:

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

Logica propozițională PL

- O propoziție este un enunț care poate fi adevărat (1) sau fals (0).
- Propozițiile sunt notate simbolic (φ, ψ, χ, $\cdots$) și sunt combinate cu ajutorul conectorilor logici ($\neg$, $\rightarrow$, $\vee$, $\wedge$, $\leftrightarrow$).

Example

Fie φ propoziția:

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

Cine este $\neg\varphi$?

Logica propozițională PL

- O propoziție este un enunț care poate fi adevărat (1) sau fals (0).
- Propozițiile sunt notate simbolic (φ, ψ, χ, $\cdots$) și sunt combinate cu ajutorul conectorilor logici ($\neg$, $\to$, $\vee$, $\wedge$, $\leftrightarrow$).

Example

Fie φ propoziția:

$$(\mathtt{stark} \wedge \neg\, \mathtt{dead}) \to (\mathtt{sansa} \vee \mathtt{arya} \vee \mathtt{bran})$$

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

$$\mathtt{stark} \wedge \neg\, \mathtt{dead} \wedge \neg \mathtt{sansa} \wedge \neg \mathtt{arya} \wedge \neg \mathtt{bran}$$

Limbajul și formulele PL

- Limbajul PL
  - variabile propoziționale: $Var = \{p, q, v, \ldots\}$
  - conectori logici: $\neg$ (unar), $\to$, $\wedge$, $\vee$, $\leftrightarrow$ (binari)
- Formulele PL

$$
\begin{array}{lcl}
var & ::= & p \mid q \mid v \mid \ldots \\
form & ::= & var \mid (\neg form) \mid form \wedge form \mid form \vee form \\
 & & \mid form \to form \mid form \leftrightarrow form
\end{array}
$$

Limbajul și formulele PL

- Limbajul PL
  - variabile propoziționale: $Var = \{p, q, v, \ldots\}$
  - conectori logici: $\neg$ (unar), $\to$, $\wedge$, $\vee$, $\leftrightarrow$ (binari)
- Formulele PL

$$
\begin{array}{lcl}
var & ::= & p \mid q \mid v \mid \ldots \\
form & ::= & var \mid (\neg form) \mid form \wedge form \mid form \vee form \\
 & & \mid form \to form \mid form \leftrightarrow form
\end{array}
$$

Example

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

Limbajul și formulele PL

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

- Formulele PL

$$
\begin{array}{lcl}
var & ::= & p \mid q \mid v \mid \ldots \\
form & ::= & var \mid (\neg form) \mid form \wedge form \mid form \vee form \\
 & & \mid form \rightarrow form \mid form \leftrightarrow form
\end{array}
$$

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

- Formulele PL

$$
\begin{array}{lcl}
var & ::= & p \mid q \mid v \mid \ldots \\
form & ::= & var \mid (\neg form) \mid form \wedge form \mid form \vee form \\
 & & \mid form \rightarrow form \mid form \leftrightarrow form
\end{array}
$$

- Conectorii sunt împărțiți în conectori de bază și conectori derivați (în funcție de formalism).

- Legături între conectori:

$$
\begin{array}{rcl}
\varphi \vee \psi & := & \neg\varphi \rightarrow \psi \\
\varphi \wedge \psi & := & \neg(\varphi \rightarrow \neg\psi) \\
\varphi \leftrightarrow \psi & := & (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)
\end{array}
$$

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`

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`

$\{(p \land \neg q) \to r, p, \neg r\} \models q$

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

$x \vee y := max\{x, y\}$ $\qquad$ $x \wedge y := min\{x, y\}$

Semantica PL

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

Semantica PL

- o funcție $e : Var \to \{0, 1\}$ se numește evaluare (interpretare)
- pentru orice evaluare $e : Var \to \{0, 1\}$ există o unică funcție $e^+ : Form \to \{0, 1\}$ care verifică următoarele proprietăți:

  - $e^+(v) = e(v)$
  - $e^+(\neg\varphi) = \neg e^+(\varphi)$
  - $e^+(\varphi \to \psi) = e^+(\varphi) \to e^+(\psi)$
  - $e^+(\varphi \wedge \psi) = e^+(\varphi) \wedge e^+(\psi)$
  - $e^+(\varphi \vee \psi) = e^+(\varphi) \vee e^+(\psi)$

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

Semantica PL

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

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

Example

Dacă $e(p) = 0$ și $e(q) = 1$ atunci

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

Semantica PL

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

Semantica PL

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

- O evaluare $e : Var \to \{0, 1\}$ 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 $e : Var \to \{0, 1\}$ 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 $e : Var \to \{0, 1\}$ 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 $e : Var \to \{0, 1\}$.
  Notăm prin $\models \varphi$ faptul că φ este o tautologie.

Semantica PL

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

- O evaluare $e : Var \to \{0,1\}$ 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 $e : Var \to \{0,1\}$.
  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 $e : Var \to \{0,1\}$.
  Notăm prin $\Gamma \models \varphi$ faptul că φ este o Γ-tautologie.

Semantica PL

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

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

Semantica PL

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

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

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

Fiecare evaluare corespunde unei linii din tabel!

Semantica PL

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

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

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

Fiecare evaluare corespunde unei linii din tabel!

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

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 conține n variabile, tabelul de adevăr are 2^n rânduri. Această metodă este atât de costisitoare computațional, încât este irealizabilă. (Timp exponențial)

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 exponențial)
- 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 exponențial)

- 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 exponențial)

- 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
- Deducția naturală
- Sistemul Gentzen

Sistemul Hilbert

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

  (A1) $\varphi \to (\psi \to \varphi)$

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

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

- Regula de deducție este modus ponens: $\dfrac{\varphi, \ \varphi \to \psi}{\psi}\,\mathbf{MP}$

Sistemul Hilbert

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

  (A1) $\varphi \to (\psi \to \varphi)$
  (A2) $(\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi))$
  (A3) $(\neg\psi \to \neg\varphi) \to (\varphi \to \psi)$.

- Regula de deducție este modus ponens: $\dfrac{\varphi, \ \varphi \to \psi}{\psi}\,\text{MP}$

- O demonstrație pentru φ este o secvență de formule $\gamma_1, \ldots, \gamma_n$ astfel încât $\gamma_n = \varphi$ și, pentru fiecare $i \in \{1, \ldots, n\}$, 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 $\gamma_j = \gamma_k \to \gamma_i$

Sistemul Hilbert

- Oricare ar fi $\varphi, \psi, \chi \in Form$ următoarele formule sunt axiome:
  - (A1) $\varphi \to (\psi \to \varphi)$
  - (A2) $(\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi))$
  - (A3) $(\neg\psi \to \neg\varphi) \to (\varphi \to \psi)$.
- Regula de deducție este modus ponens: $\dfrac{\varphi, \ \varphi \to \psi}{\psi}$ MP
- O demonstrație pentru φ este o secvență de formule $\gamma_1, \ldots, \gamma_n$ astfel încât $\gamma_n = \varphi$ și, pentru fiecare $i \in \{1, \ldots, n\}$, 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 $\gamma_j = \gamma_k \to \gamma_i$
- 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: $\dfrac{\varphi, \ \varphi \rightarrow \psi}{\psi}$ MP

Example

Fie φ si 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: $\dfrac{\varphi,\ \varphi \rightarrow \psi}{\psi}\,\mathrm{MP}$

Example

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

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

Avem următoarea demonstrație:

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

Sistemul Hilbert

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

  (A1) $\varphi \to (\psi \to \varphi)$
  (A2) $(\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi))$
  (A3) $(\neg\psi \to \neg\varphi) \to (\varphi \to \psi)$.

- Regula de deducție este modus ponens: $\dfrac{\varphi, \ \varphi \to \psi}{\psi}\,\textbf{MP}$

Teorema de completitudine

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

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

($\Rightarrow$) Corectitudine
($\Leftarrow$) Completitudine

PL - Deducție naturala

Deducția naturală[1] pe scurt

- În deducția naturală deducem (demonstrăm) formule din alte formule folosind reguli de deducție.

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

Deducția naturală[1] pe scurt

- În deducția naturală deducem (demonstrăm) formule din alte formule folosind reguli de deducție.

- Numim secvent o expresie de forma

$$\varphi_1, \ldots, \varphi_n \vdash \psi$$

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

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

Deducția naturală[1] pe scurt

- În deducția naturală deducem (demonstrăm) formule din alte formule folosind reguli de deducție.
- Numim secvent o expresie de forma
$$\varphi_1, \ldots, \varphi_n \vdash \psi$$
Formulele $\varphi_1, \ldots, \varphi_n$ se numesc premise, iar ψ se numește concluzie.
- Un secvent este valid dacă există o demonstrație folosind regulile de deducție.

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

Deducția naturală[1] pe scurt

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

$$\varphi_1, \ldots, \varphi_n \vdash \psi$$

  Formulele $\varphi_1, \ldots, \varphi_n$ 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).

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

Deducția naturală[1] pe scurt

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

$$\varphi_1, \ldots, \varphi_n \vdash \psi$$

  Formulele $\varphi_1, \ldots, \varphi_n$ 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.

[1] M. 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

$$\frac{\varphi \qquad \psi}{\varphi \wedge \psi}\ (\wedge i)$$

  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

$$\frac{\varphi \qquad \psi}{\varphi \wedge \psi}\ (\wedge i)$$

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

- Regulile pentru $\wedge$- eliminare sunt:

$$\frac{\varphi \wedge \psi}{\varphi}\ (\wedge e_1) \qquad \frac{\varphi \wedge \psi}{\psi}\ (\wedge e_2)$$

Regulile pentru conjuncție

Example

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

Regulile pentru conjuncție

Example

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

Putem scrie demonstrația ca un arbore

$$\dfrac{\dfrac{p \land q}{q}\,(\land e_2) \qquad r}{q \land r}\,(\land i)$$

Regulile pentru conjuncție

Example

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

Putem scrie demonstrația ca un arbore

$$\dfrac{\dfrac{p \wedge q}{q}\,(\wedge e_2) \qquad r}{q \wedge r}\,(\wedge i)$$

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

$$\dfrac{\dfrac{p \wedge q}{q}\,(\wedge e_2) \qquad r}{q \wedge r}\,(\wedge i)$$

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

$$\dfrac{\dfrac{p \wedge q}{q}\,(\wedge e_2) \qquad r}{q \wedge r}\,(\wedge i)$$

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:

$$\frac{\neg\neg\varphi}{\varphi}\ (\neg\neg e) \qquad \frac{\varphi}{\neg\neg\varphi}\ (\neg\neg i)$$

Regulile pentru dubla negație

- Regulile ¬¬-introducere și ¬¬-eliminare sunt:

$$\frac{\neg\neg\varphi}{\varphi}\ (\neg\neg e) \qquad \frac{\varphi}{\neg\neg\varphi}\ (\neg\neg i)$$

Example

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

Regulile pentru dubla negație

- Regulile ¬¬-introducere și ¬¬-eliminare sunt:

$$\frac{\neg\neg\varphi}{\varphi}\ (\neg\neg e) \qquad \frac{\varphi}{\neg\neg\varphi}\ (\neg\neg i)$$

Example

Demonstrați că secventul $\neg\neg(q \land r) \vdash \neg\neg r$ este valid.

1	$\neg\neg(q \land r)$	*premisa*
2	$q \land r$	$(\neg\neg ei)$,1
3	r	$(\land e_2)$,2

Regulile pentru dubla negație

- Regulile ¬¬-introducere și ¬¬-eliminare sunt:

$$\frac{\neg\neg\varphi}{\varphi}\ (\neg\neg e) \qquad \frac{\varphi}{\neg\neg\varphi}\ (\neg\neg i)$$

Example

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

1	$\neg\neg(q \wedge r)$	*premisa*
2	$q \wedge r$	$(\neg\neg ei)$,1
3	r	$(\wedge e_2)$,2
4	$\neg\neg r$	$(\neg\neg i)$,3

Regulile pentru implicație: $\rightarrow$-eliminare

- Regula de $\rightarrow$-eliminare o stiți deja:

Regulile pentru implicație: →-eliminare

- Regula de →-eliminare o stiți deja: este modus ponens:

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

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.
  Acest lucru se reprezintă astfel:

$$\frac{\boxed{\begin{array}{c}\varphi\\ \vdots\\ \psi\end{array}}}{\varphi \rightarrow \psi}\ (\rightarrow i)$$

Regulile pentru implicație: $\to$-introducere

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

$$\frac{\boxed{\begin{array}{c}\varphi \\ \vdots \\ \psi\end{array}}}{\varphi \to \psi}\;(\to i)$$

- 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 $\varphi \to \psi$ 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

$$\boxed{\dfrac{p \wedge q}{p}\ (\wedge e_1)}$$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \wedge q) \rightarrow p$

Vom considera $p \wedge q$ ca ipoteză temporară

$$\boxed{\dfrac{p \wedge q}{p}\ (\wedge e_1)}$$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \wedge q) \rightarrow p$

Vom considera $p \wedge q$ ca ipoteză temporară

$$\dfrac{\boxed{\dfrac{p \wedge q}{p}\ (\wedge e_1)}}{p \wedge q \rightarrow p}\ (\rightarrow i)$$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \wedge q) \rightarrow p$

Putem scrie demonstrația într-un mod liniar în felul următor:

1	$p \wedge q$	*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 $\vdash (p \land q) \rightarrow p$

Putem scrie demonstrația într-un mod liniar în felul următor:

$$
\begin{array}{lll}
1 & \boxed{\begin{array}{ll} p \land q & \textit{ipoteza} \\ p & (\land e_1),1 \end{array}} & \\
2 & & \\
3 & p \land q \rightarrow p & (\rightarrow i),1\text{–}2
\end{array}
$$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash p \to p$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash p \rightarrow p$

1	$\boxed{p}$	*ipoteza*
2	$p \rightarrow p$	$(\rightarrow i)$,1

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \to q) \to ((q \to r) \to (p \to r))$

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \to q) \to ((q \to r) \to (p \to r))$

1	$p \to q$	*ipoteza*
2	$q \to r$	*ipoteza*
3	p	*ipoteza*

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \to q) \to ((q \to r) \to (p \to r))$

1	$p \to q$	*ipoteza*
2	$q \to r$	*ipoteza*
3	p	*ipoteza*
4	q	$(\to e)$,1,3
5	r	$(\to e)$,2,4

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \to q) \to ((q \to r) \to (p \to r))$

1	$p \to q$	*ipoteza*
2	$q \to r$	*ipoteza*
3	p	*ipoteza*
4	q	$(\to e)$,1,3
5	r	$(\to e)$,2,4
6	$p \to r$	$(\to i)$,3–5

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \to q) \to ((q \to r) \to (p \to r))$

1	$p \to q$	*ipoteza*
2	$q \to r$	*ipoteza*
3	p	*ipoteza*
4	q	$(\to e)$,1,3
5	r	$(\to e)$,2,4
6	$p \to r$	$(\to i)$,3–5
7	$(q \to r) \to (p \to r)$	$(\to i)$,2–6

Regulile pentru implicație

Example

Demonstrați teorema $\vdash (p \to q) \to ((q \to r) \to (p \to r))$

1	$p \to q$	*ipoteza*
2	$q \to r$	*ipoteza*
3	p	*ipoteza*
4	q	$(\to e)$,1,3
5	r	$(\to e)$,2,4
6	$p \to r$	$(\to i)$,3–5
7	$(q \to r) \to (p \to r)$	$(\to i)$,2–6
8	$(p \to q) \to ((q \to r) \to (p \to r))$	$(\to i)$,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 $\vdash p \to (q \to p)$

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 $\vdash p \to (q \to p)$

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 $\vdash p \to (q \to p)$

1	p	*ipoteza*
2	q	*ipoteza*
3	p	*copiere* 1
4	$q \to p$	$(\to i)$,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 $\vdash p \to (q \to p)$

1	p	*ipoteza*
2	q	*ipoteza*
3	p	*copiere* 1
4	$q \to p$	$(\to i)$,2–3
5	$p \to (q \to p)$	$(\to i)$,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

$$\frac{\varphi}{\varphi \vee \psi}\ (\vee i_1) \qquad \frac{\psi}{\varphi \vee \psi}\ (\vee i_2)$$

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

$$\frac{\varphi}{\varphi \vee \psi}\ (\vee i_1) \qquad \frac{\psi}{\varphi \vee \psi}\ (\vee i_2)$$

Example

Demonstrați că secventul $q \to r \vdash q \to (r \vee p)$ este valid.

Regulile pentru disjuncție: ∨-introducere

- Intuitiv, a demonstra $\varphi \vee \psi$ revine la a demonstra φ sau ψ. În consecință, regulile de ∨-introducere sunt

$$\frac{\varphi}{\varphi \vee \psi}\ (\vee i_1) \qquad \frac{\psi}{\varphi \vee \psi}\ (\vee i_2)$$

Example

Demonstrați că secventul $q \to r \vdash q \to (r \vee p)$ este valid.

1	$q \to r$	*premisa*
2	q	*ipoteza*
3	r	$(\to e)$,1,2
4	$r \vee p$	$(\vee i_1)$,3
5	$q \to (r \vee p)$	$(\to i)$,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: ∨-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 ∨-eliminare reflectă aceast argument:

$$\frac{\varphi \vee \psi \quad \boxed{\begin{array}{c}\varphi \\ \vdots \\ \chi\end{array}} \quad \boxed{\begin{array}{c}\psi \\ \vdots \\ \chi\end{array}}}{\chi}\ (\vee e)$$

Regulile pentru disjuncție

Example

Demonstrați că secventul $q \to r \vdash (p \vee q) \to (p \vee r)$ este valid.

1	$q \to r$	*premisa*
2	$p \vee q$	*ipoteza*
3	p	*ipoteza*
4	$p \vee r$	$(\vee i_1)$,3
5	q	*ipoteza*
6	r	$(\to e)$,1,5
7	$p \vee r$	$(\vee i_2)$,6
8	$p \vee r$	$(\vee e)$,2,3–4,5–7
9	$p \vee q \to p \vee r$	$(\to i)$,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:

$$\frac{\varphi \quad \neg\varphi}{\perp}\ (\neg e) \qquad\qquad \frac{\boxed{\begin{array}{c}\varphi\\ \vdots\\ \perp\end{array}}}{\neg\varphi}\ (\neg i)$$

Regulile pentru negație

Example

Demonstrați că secventul $p \to \neg p \vdash \neg p$ este valid.

1	$p \to \neg p$	*premisa*
2	p	*ipoteza*
3	$\neg p$	$(\to e)$,1,2
4	$\bot$	$(\neg e)$,2,3
5	$\neg p$	$(\neg i)$,2–4

Regulile DN

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi}\ (\wedge i) \qquad \frac{\varphi \wedge \psi}{\varphi}\ (\wedge e_1) \qquad \frac{\varphi \wedge \psi}{\psi}\ (\wedge e_2)$$

$$\frac{\boxed{\begin{array}{c}\varphi\\ \vdots\\ \psi\end{array}}}{\varphi \to \psi}\ (\to i) \qquad \frac{\varphi \quad \varphi \to \psi}{\psi}\ (\to e)$$

$$\frac{\varphi}{\varphi \vee \psi}\ (\vee i_1) \qquad \frac{\psi}{\varphi \vee \psi}\ (\vee i_2) \qquad \frac{\varphi \vee \psi \quad \boxed{\begin{array}{c}\varphi\\ \vdots\\ \chi\end{array}} \quad \boxed{\begin{array}{c}\psi\\ \vdots\\ \chi\end{array}}}{\chi}\ (\vee e)$$

$$\frac{\varphi}{\neg\neg\varphi}\ (\neg\neg i) \qquad \frac{\boxed{\begin{array}{c}\varphi\\ \vdots\\ \bot\end{array}}}{\neg\varphi}\ (\neg i) \qquad \frac{\neg\neg\varphi}{\varphi}\ (\neg\neg e) \qquad \frac{\varphi \quad \neg\varphi}{\bot}\ (\neg e) \qquad \frac{\bot}{\varphi}\ (\bot e)$$

Reguli derivate

- Următoarele reguli pot fi derivate din regulile deducției naturale:

$$\frac{\varphi \to \psi \quad \neg\psi}{\neg\varphi}\ \textbf{MT} \qquad \frac{\boxed{\begin{array}{c}\neg\varphi\\ \vdots\\ \bot\end{array}}}{\varphi}\ \textbf{RAA} \qquad \frac{}{\varphi \vee \neg\varphi}\ \textbf{TND}$$

Deducția 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 naturala: Corectitudinea

Corectitudinea DN

Teoremă

Deducția naturală este corectă, i.e.

$$\text{dacă } \varphi_1, \ldots, \varphi_n \vdash \varphi \text{ este valid atunci } \varphi_1, \ldots, \varphi_n \models \varphi$$

oricare ar fi $n \geq 0$ și formulele $\varphi_1, \ldots, \varphi_n$, φ.

Corectitudinea DN

Teoremă

Deducția naturală este corectă, i.e.

$$\text{dacă } \varphi_1, \ldots, \varphi_n \vdash \varphi \text{ este valid atunci } \varphi_1, \ldots, \varphi_n \models \varphi$$

oricare ar fi $n \geq 0$ și formulele $\varphi_1, \ldots, \varphi_n$, φ.

Demonstrație

Din ipoteză știm că există o demonstrație pentru φ din ipotezele $\varphi_1, \ldots, \varphi_n$ folosind regulile deducției naturale.

Corectitudinea DN

Teoremă

Deducția naturală este corectă, i.e.

$$\text{dacă } \varphi_1, \ldots, \varphi_n \vdash \varphi \text{ este valid atunci } \varphi_1, \ldots, \varphi_n \models \varphi$$

oricare ar fi $n \geq 0$ și formulele $\varphi_1, \ldots, \varphi_n$, φ.

Demonstrație

Din ipoteză știm că există o demonstrație pentru φ din ipotezele $\varphi_1, \ldots, \varphi_n$ folosind regulile deducției naturale.

Fie k numărul de linii dintr-o demonstrație în forma liniară.

Corectitudinea DN

Teoremă

Deducția naturală este corectă, i.e.

$$\text{dacă } \varphi_1, \ldots, \varphi_n \vdash \varphi \text{ este valid atunci } \varphi_1, \ldots, \varphi_n \models \varphi$$

oricare ar fi $n \geq 0$ și formulele $\varphi_1, \ldots, \varphi_n$, φ.

Demonstrație

Din ipoteză știm că există o demonstrație pentru φ din ipotezele $\varphi_1, \ldots, \varphi_n$ 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 $\varphi_1, \cdots, \varphi_n$, φ formule, dacă $\varphi_1, \ldots, \varphi_n \vdash \varphi$ are o demonstrație de lungime $k \geq 1$ atunci $\varphi_1, \ldots, \varphi_n \models \varphi$,

(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 secventul inițial este $\varphi \vdash \varphi$.

Este evident că $\varphi \models \varphi$

Corectitudinea DN

Demonstrație (cont.)

<u>Cazul de inducție.</u> Vom presupune că:

<u>oricare ar fi $\varphi_1, \cdots, \varphi_n$, φ, dacă $\varphi_1, \ldots, \varphi_n \vdash \varphi$ are o demonstrație de lungime $< k$ atunci $\varphi_1, \ldots, \varphi_n \models \varphi$</u>

și vom demonstra că proprietatea este adevărată pentru secvenți cu demonstrații de lungime k.

Corectitudinea DN

Demonstrație (cont.)

Cazul de inducție. Vom presupune că:

oricare ar fi $\varphi_1, \cdots, \varphi_n$, φ, dacă $\varphi_1, \ldots, \varphi_n \vdash \varphi$ are o demonstrație de lungime $< k$ atunci $\varphi_1, \ldots, \varphi_n \models \varphi$

și vom demonstra că proprietatea este adevărată pentru secvenți cu demonstrații de lungime k.

Fie (R) ultima regulă care se aplică în demonstrație, adică

1	φ_1	*premisa*
	$\vdots$	
n	φ_n	*premisa*
	$\vdots$	
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*
	$\vdots$	
n	φ_n	*premisa*
	$\vdots$	
k_1	ψ	
	$\vdots$	
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*
	$\vdots$	
n	φ_n	*premisa*
	$\vdots$	
k_1	ψ	
	$\vdots$	
k_2	χ	
k	$\psi \wedge \chi$	$(\wedge i) k_1, k_2$

Se observă că secvenții
$\varphi_1, \ldots, \varphi_n \vdash \psi$ și
$\varphi_1, \ldots, \varphi_n \vdash \chi$
au demonstrații de lungime $< k$.

Corectitudinea DN

Demonstrație (cont.)

Presupunem că ultima regulă a fost $(\wedge i)$. Aceasta înseamnă că

$$\varphi = \psi \wedge \chi$$

1	φ_1	*premisa*
	$\vdots$	
n	φ_n	*premisa*
	$\vdots$	
k_1	ψ	
	$\vdots$	
k_2	χ	
k	$\psi \wedge \chi$	$(\wedge i) k_1, k_2$

Se observă că secvențiii
$\varphi_1, \ldots, \varphi_n \vdash \psi$ și
$\varphi_1, \ldots, \varphi_n \vdash \chi$
au demonstrații de lungime $< k$.

Din ipoteza de inducție rezultă
$\varphi_1, \ldots, \varphi_n \models \psi$ și
$\varphi_1, \ldots, \varphi_n \models \chi$

Corectitudinea DN

Demonstrație (cont.)

Presupunem că ultima regulă a fost $(\wedge i)$. Aceasta înseamnă că

$$\varphi = \psi \wedge \chi$$

1	φ_1	*premisa*
	$\vdots$	
n	φ_n	*premisa*
	$\vdots$	
k_1	ψ	
	$\vdots$	
k_2	χ	
k	$\psi \wedge \chi$	$(\wedge i) k_1, k_2$

Se observă că secvenții
$\varphi_1, \ldots, \varphi_n \vdash \psi$ și
$\varphi_1, \ldots, \varphi_n \vdash \chi$
au demonstrații de lungime $< k$.

Din ipoteza de inducție rezultă
$\varphi_1, \ldots, \varphi_n \models \psi$ și
$\varphi_1, \ldots, \varphi_n \models \chi$ deci

$\varphi_1, \ldots, \varphi_n \models \psi \wedge \chi$

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.

$$\begin{array}{lll}
1 & \varphi_1 & \textit{premisa} \\
 & \vdots & \\
n & \varphi_n & \textit{premisa} \\
 & \vdots & \\
k_1 & \boxed{\begin{array}{ll} \psi & \textit{ipoteza} \\ \vdots & \\ \chi & \end{array}} & \\
 & & \\
k & \psi \rightarrow \chi & (\rightarrow i)\, k_1 - k_2
\end{array}$$

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*
	$\vdots$	
n	φ_n	*premisa*
	$\vdots$	
k_1	ψ	*ipoteza*
	$\vdots$	
k_2	χ	
k	$\psi \rightarrow \chi$	$(\rightarrow i) k_1 - k_2$

Se observă că

$$\varphi_1, \ldots, \varphi_n, \psi \vdash \chi$$

are demonstrația de lungime $< k$.

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*
	$\vdots$	
n	φ_n	*premisa*
	$\vdots$	
k_1	ψ	*ipoteza*
	$\vdots$	
k_2	χ	
k	$\psi \rightarrow \chi$	$(\rightarrow i) k_1 - k_2$

Se observă că

$$\varphi_1, \ldots, \varphi_n, \psi \vdash \chi$$

are demonstrația de lungime $< k$.

Din ipoteza de inducție rezultă

$$\varphi_1, \ldots, \varphi_n, \psi \models \chi \quad (*)$$

Corectitudinea DN

Demonstrație (cont.)

Putem acum să demonstrăm că $\varphi_1, \ldots, \varphi_n, \models \varphi$.

Corectitudinea DN

Demonstrație (cont.)

Putem acum să demonstrăm că $\varphi_1, \ldots, \varphi_n, \models \varphi$.

Fie $e : Var \to \{0, 1\}$ o evaluare astfel încât $e^+(\varphi_1) = \cdots = e^+(\varphi_n) = 1$. Vrem să arătăm că $e^+(\varphi) = 1$.

Deoarece $\varphi = \psi \to \chi$ considerăm două cazuri.

Dacă $e^+(\psi) = 0$ atunci $e^+(\varphi) = 0 \to e^+(\chi) = 1$.

Corectitudinea DN

Demonstrație (cont.)

Putem acum să demonstrăm că $\varphi_1, \ldots, \varphi_n, \models \varphi$.

Fie $e : Var \to \{0, 1\}$ o evaluare astfel încât $e^+(\varphi_1) = \cdots = e^+(\varphi_n) = 1$. Vrem să arătăm că $e^+(\varphi) = 1$.

Deoarece $\varphi = \psi \to \chi$ considerăm două cazuri.

Dacă $e^+(\psi) = 0$ atunci $e^+(\varphi) = 0 \to e^+(\chi) = 1$.

Dacă $e^+(\psi) = 1$ atunci e^+ este un model pentru formulele $\varphi_1, \ldots, \varphi_n, \psi$. Din $(*)$ rezultă ca $e^+(\chi) = 1$, deci $e^+(\varphi) = 1 \to 1 = 1$.

Corectitudinea DN

Demonstrație (cont.)

Putem acum să demonstrăm că $\varphi_1, \ldots, \varphi_n, \models \varphi$.

Fie $e : Var \to \{0, 1\}$ o evaluare astfel încât $e^+(\varphi_1) = \cdots = e^+(\varphi_n) = 1$.
Vrem să arătăm că $e^+(\varphi) = 1$.

Deoarece $\varphi = \psi \to \chi$ considerăm două cazuri.

Dacă $e^+(\psi) = 0$ atunci $e^+(\varphi) = 0 \to e^+(\chi) = 1$.

Dacă $e^+(\psi) = 1$ atunci e^+ este un model pentru formulele $\varphi_1, \ldots, \varphi_n, \psi$.
Din $(*)$ rezultă ca $e^+(\chi) = 1$, deci $e^+(\varphi) = 1 \to 1 = 1$.

Am demonstrat că regula $(\to i)$ este corectă.

Corectitudinea DN

Demonstrație (cont.)

Putem acum să demonstrăm că $\varphi_1, \ldots, \varphi_n, \models \varphi$.

Fie $e : Var \to \{0, 1\}$ o evaluare astfel încât $e^+(\varphi_1) = \cdots = e^+(\varphi_n) = 1$.
Vrem să arătăm că $e^+(\varphi) = 1$.

Deoarece $\varphi = \psi \to \chi$ considerăm două cazuri.

Dacă $e^+(\psi) = 0$ atunci $e^+(\varphi) = 0 \to e^+(\chi) = 1$.

Dacă $e^+(\psi) = 1$ atunci e^+ este un model pentru formulele $\varphi_1, \ldots, \varphi_n, \psi$.
Din $(*)$ rezultă ca $e^+(\chi) = 1$, deci $e^+(\varphi) = 1 \to 1 = 1$.

Am demonstrat că regula $(\to i)$ este corectă.

Pentru a finaliza demonstrația trebuie sa arătăm că fiecare din celelalte reguli ale deducției naturale este corectă. $\square$

PL - Deducție naturală: Completitudinea (opțional)

Completitudinea DN (opțional)

Teoremă

Deducția naturală este completă, i.e.

$$\text{dacă } \varphi_1, \ldots, \varphi_n \models \varphi \ \text{ atunci } \ \varphi_1, \ldots, \varphi_n \vdash \varphi \text{ este valid}$$

oricare ar fi formulele $\varphi_1, \ldots, \varphi_n$, φ.

Notații

Pentru a demonstra ca DN este completă pentru PL facem urmatoarele notații:

Notații

Pentru a demonstra ca DN este completă pentru PL facem urmatoarele notații:

- Fie $e : Var \to \{0, 1\}$ evaluare. Pentru orice $v \in Var$ definim

$$v^e := \begin{cases} v & \text{dacă } e(v) = 1 \\ \neg v & \text{dacă } e(v) = 0 \end{cases}$$

- $Var(\varphi) := \{v \in Var \mid v \text{ apare în } \varphi\}$ oricare φ formulă.

Completitudinea DN - rezultate ajutătoare

Propozitia 1

Fie φ este o formulă si $Var(\varphi) = \{v_1, \ldots, v_n\}$. Pentru orice evaluare $e : Var \to \{0, 1\}$ sunt adevarate:

- $e^+(\varphi) = 1$ implica $\{v_1^e, \ldots, v_n^e\} \vdash \varphi$ este valid,
- $e^+(\varphi) = 0$ implica $\{v_1^e, \ldots, v_n^e\} \vdash \neg\varphi$ este valid.

Completitudinea DN - rezultate ajutătoare

Propozitia 1

Fie φ este o formulă si $Var(\varphi) = \{v_1, \ldots, v_n\}$. Pentru orice evaluare $e : Var \to \{0, 1\}$ sunt adevarate:

- $e^+(\varphi) = 1$ implica $\{v_1^e, \ldots, v_n^e\} \vdash \varphi$ este valid,
- $e^+(\varphi) = 0$ implica $\{v_1^e, \ldots, v_n^e\} \vdash \neg\varphi$ este valid.

Propozitia 2

Oricare ar fi formulele $\varphi_1, \ldots, \varphi_n, \varphi$,
daca $\varphi_1, \ldots, \varphi_n \models \varphi$ atunci $\models \varphi_1 \to (\varphi_2 \to (\cdots \to (\varphi_n \to \varphi) \cdots))$.

Completitudinea DN - rezultate ajutătoare

Propozitia 1

Fie φ este o formulă si $Var(\varphi) = \{v_1, \ldots, v_n\}$. Pentru orice evaluare $e : Var \to \{0, 1\}$ sunt adevarate:

- $e^+(\varphi) = 1$ implica $\{v_1^e, \ldots, v_n^e\} \vdash \varphi$ este valid,
- $e^+(\varphi) = 0$ implica $\{v_1^e, \ldots, v_n^e\} \vdash \neg\varphi$ este valid.

Propozitia 2

Oricare ar fi formulele $\varphi_1, \ldots, \varphi_n$, φ,
daca $\varphi_1, \ldots, \varphi_n \models \varphi$ atunci $\models \varphi_1 \to (\varphi_2 \to (\cdots \to (\varphi_n \to \varphi) \cdots))$.

Propozitia 3

Oricare ar fi formulele $\varphi_1, \ldots, \varphi_n$, φ,
dacă $\vdash \varphi_1 \to (\varphi_2 \to (\cdots \to (\varphi_n \to \varphi) \cdots))$ este valid,
atunci $\varphi_1, \ldots, \varphi_n \vdash \varphi$ este valid.

Completitudinea DN

Teoremă

Deducția naturală este completă, i.e.

$$\text{dacă } \varphi_1, \ldots, \varphi_n \models \varphi \quad \text{atunci} \quad \varphi_1, \ldots, \varphi_n \vdash \varphi \text{ este valid}$$

oricare ar fi formulele $\varphi_1, \ldots, \varphi_n$, φ.

Completitudinea DN

Teoremă

Deducția naturală este completă, i.e.

$$\text{dacă } \varphi_1, \ldots, \varphi_n \models \varphi \text{ atunci } \varphi_1, \ldots, \varphi_n \vdash \varphi \text{ este valid}$$

oricare ar fi formulele $\varphi_1, \ldots, \varphi_n$, φ.

Demonstratie

Pasul 1. Daca $\models \varphi$ atunci $\vdash \varphi$ este valid.

Completitudinea DN

Teoremă

Deducția naturală este completă, i.e.

$$\text{dacă } \varphi_1, \ldots, \varphi_n \models \varphi \text{ atunci } \varphi_1, \ldots, \varphi_n \vdash \varphi \text{ este valid}$$

oricare ar fi formulele $\varphi_1, \ldots, \varphi_n$, φ.

Demonstratie

Pasul 1. Daca $\models \varphi$ atunci $\vdash \varphi$ este valid.

Pasul 2. Presupunem ca $\varphi_1, \ldots, \varphi_n \models \varphi$.

Completitudinea DN

Teoremă

Deducția naturală este completă, i.e.

$$\text{dacă } \varphi_1, \ldots, \varphi_n \models \varphi \ \text{ atunci } \ \varphi_1, \ldots, \varphi_n \vdash \varphi \text{ este valid}$$

oricare ar fi formulele $\varphi_1, \ldots, \varphi_n$, φ.

Demonstratie

Pasul 1. Daca $\models \varphi$ atunci $\vdash \varphi$ este valid.

Pasul 2. Presupunem ca $\varphi_1, \ldots, \varphi_n \models \varphi$.
Din Propozitia 2 deducem ca $\models \varphi_1 \to (\varphi_2 \to (\cdots \to (\varphi_n \to \varphi) \cdots))$.

Completitudinea DN

Teoremă

Deducția naturală este completă, i.e.

$$\text{dacă } \varphi_1, \ldots, \varphi_n \models \varphi \text{ atunci } \varphi_1, \ldots, \varphi_n \vdash \varphi \text{ este valid}$$

oricare ar fi formulele $\varphi_1, \ldots, \varphi_n$, φ.

Demonstratie

Pasul 1. Daca $\models \varphi$ atunci $\vdash \varphi$ este valid.

Pasul 2. Presupunem ca $\varphi_1, \ldots, \varphi_n \models \varphi$.
Din Propozitia 2 deducem ca $\models \varphi_1 \to (\varphi_2 \to (\cdots \to (\varphi_n \to \varphi) \cdots))$.
Aplicand Pasul 1 obtinem ca $\vdash \varphi_1 \to (\varphi_2 \to (\cdots \to (\varphi_n \to \varphi) \cdots))$ este valid.

Completitudinea DN

Teoremă

Deducția naturală este completă, i.e.

$$\text{dacă } \varphi_1, \ldots, \varphi_n \models \varphi \quad \text{atunci} \quad \varphi_1, \ldots, \varphi_n \vdash \varphi \text{ este valid}$$

oricare ar fi formulele $\varphi_1, \ldots, \varphi_n$, φ.

Demonstratie

Pasul 1. Daca $\models \varphi$ atunci $\vdash \varphi$ este valid.

Pasul 2. Presupunem ca $\varphi_1, \ldots, \varphi_n \models \varphi$.
Din Propozitia 2 deducem ca $\models \varphi_1 \to (\varphi_2 \to (\cdots \to (\varphi_n \to \varphi) \cdots))$.
Aplicand Pasul 1 obtinem ca $\vdash \varphi_1 \to (\varphi_2 \to (\cdots \to (\varphi_n \to \varphi) \cdots))$ este valid. In consecinta $\varphi_1, \ldots, \varphi_n \vdash \varphi$ este valid din Propozitia 3.

Completitudinea DN

Demonstratie (cont.)

În continuare demonstram Pasul 1.

Fie φ o tautologie, i.e. $\models \varphi$, astfel incat $Var(\varphi) = \{p_1, \ldots, p_n\}$.

Completitudinea DN

Demonstratie (cont.)

În continuare demonstram Pasul 1.

Fie φ o tautologie, i.e. $\models \varphi$, astfel incat $Var(\varphi) = \{p_1, \ldots, p_n\}$.

Oricare ar fi $e : Var \to \{0, 1\}$ stim ca $e^+(\varphi) = 1$ deci, din Propozitia 1, rezulta ca secventul $\{p_1^e, \ldots, p_n^e\} \vdash \varphi$ este valid.

Completitudinea DN

Demonstratie (cont.)

În continuare demonstram Pasul 1.

Fie φ o tautologie, i.e. $\models \varphi$, astfel incat $Var(\varphi) = \{p_1, \ldots, p_n\}$.

Oricare ar fi $e : Var \to \{0, 1\}$ stim ca $e^+(\varphi) = 1$ deci, din Propozitia 1, rezulta ca secventul $\{p_1^e, \ldots, p_n^e\} \vdash \varphi$ 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.

Completitudinea DN

Demonstratie (cont.)

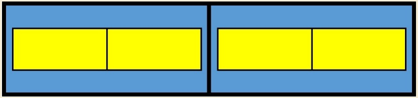
În continuare demonstram Pasul 1.

Fie φ o tautologie, i.e. $\models \varphi$, astfel incat $Var(\varphi) = \{p_1, \ldots, p_n\}$.

Oricare ar fi $e : Var \to \{0, 1\}$ stim ca $e^+(\varphi) = 1$ deci, din Propozitia 1, rezulta ca secventul $\{p_1^e, \ldots, p_n^e\} \vdash \varphi$ 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. $Var(\varphi) = \{p_1, p_2\}$.

De exemplu, puteti considera $\varphi = p_1 \wedge p_2 \to p_1$

Completitudinea DN

Demonstratie (cont.)

Consideram $\models \varphi$ si $n = 2$, i.e. $Var(\varphi) = \{p_1, p_2\}$.

De exemplu, puteti considera $\varphi = p_1 \wedge p_2 \rightarrow p_1$

Din Propozitia 1 stim ca urmatorii secventi sunt valizi:

$$\begin{aligned} p_1, p_2 &\quad \vdash \varphi \\ p_1, \neg p_2 &\quad \vdash \varphi \\ \neg p_1, p_2 &\quad \vdash \varphi \\ \neg p_1, \neg p_2 &\quad \vdash \varphi \end{aligned}$$

Completitudinea DN

Demonstratie (cont.)

Consideram $\models \varphi$ si $n = 2$, i.e. $Var(\varphi) = \{p_1, p_2\}$.

De exemplu, puteti considera $\varphi = p_1 \wedge p_2 \rightarrow p_1$

Din Propozitia 1 stim ca urmatorii secventi sunt valizi:

$$\begin{array}{rl} p_1, p_2 & \vdash \varphi \\ p_1, \neg p_2 & \vdash \varphi \\ \neg p_1, p_2 & \vdash \varphi \\ \neg p_1, \neg p_2 & \vdash \varphi \end{array}$$

deci exista demonstratiile:

$$\boxed{\begin{array}{ll} p_1 & \textit{ipoteza} \\ p_2 & \textit{ipoteza} \\ \vdots & \\ \varphi & \end{array}} \quad \boxed{\begin{array}{ll} p_1 & \textit{ipoteza} \\ \neg p_2 & \textit{ipoteza} \\ \vdots & \\ \varphi & \end{array}} \quad \boxed{\begin{array}{ll} \neg p_1 & \textit{ipoteza} \\ p_2 & \textit{ipoteza} \\ \vdots & \\ \varphi & \end{array}} \quad \boxed{\begin{array}{ll} \neg p_1 & \textit{ipoteza} \\ \neg p_2 & \textit{ipoteza} \\ \vdots & \\ \varphi & \end{array}}$$

Completitudinea DN

Demonstratie (cont.)

Combinam cele patru demonstratii astfel:

Completitudinea DN

Demonstratie (cont.)

Combinam cele patru demonstratii astfel:

$$
\begin{array}{llll}
p_1 \vee \neg p_1 & & & \textit{TND} \\
\hline
\multicolumn{1}{|l}{p_1} & \multicolumn{1}{l|}{\textit{ipoteza}} & \multicolumn{1}{|l}{\neg p_1} & \multicolumn{1}{l|}{\textit{ipoteza}}
\end{array}
$$

Completitudinea DN

Demonstratie (cont.)

Combinam cele patru demonstratii astfel:

$p_1 \vee \neg p_1$	*TND*
p_1	*ipoteza*
$p_2 \vee \neg p_2$	*TND*
p_2	*ipoteza*
$\neg p_2$	*ipoteza*
$\neg p_1$	*ipoteza*
$p_2 \vee \neg p_2$	*TND*
p_2	*ipoteza*
$\neg p_2$	*ipoteza*

Completitudinea DN

Demonstratie (cont.)

Combinam cele patru demonstratii astfel:

$p_1 \lor \neg p_1$ *TND*

p_1	*ipoteza*
$p_2 \lor \neg p_2$	*TND*
p_2	*ipoteza*
$\vdots$	
φ	
$\neg p_2$	*ipoteza*
$\vdots$	
φ	
φ	$(\lor e)$

$\neg p_1$	*ipoteza*
$p_2 \lor \neg p_2$	*TND*
p_2	*ipoteza*
$\vdots$	
φ	
$\neg p_2$	*ipoteza*
$\vdots$	
φ	
φ	$(\lor e)$

Completitudinea DN

Demonstratie (cont.)

Combinam cele patru demonstratii astfel:

$p_1 \vee \neg p_1$ — *TND*

p_1	*ipoteza*
$p_2 \vee \neg p_2$	*TND*
p_2	*ipoteza*
$\vdots$	
φ	
$\neg p_2$	*ipoteza*
$\vdots$	
φ	
φ	$(\vee e)$

$\neg p_1$	*ipoteza*
$p_2 \vee \neg p_2$	*TND*
p_2	*ipoteza*
$\vdots$	
φ	
$\neg p_2$	*ipoteza*
$\vdots$	
φ	
φ	$(\vee e)$

φ — $(\vee e)$

Completitudinea DN

Demonstratie (cont.)

Combinam cele patru demonstratii astfel:

$p_1 \vee \neg p_1$ — *TND*

p_1 — *ipoteza*	$\neg p_1$ — *ipoteza*
$p_2 \vee \neg p_2$ — *TND*	$p_2 \vee \neg p_2$ — *TND*
p_2 *ipoteza* / $\vdots$ / φ	p_2 *ipoteza* / $\vdots$ / φ
$\neg p_2$ *ipoteza* / $\vdots$ / φ	$\neg p_2$ *ipoteza* / $\vdots$ / φ
φ — $(\vee e)$	φ — $(\vee e)$

φ — $(\vee e)$

Am obtinut o demonstratie pentru φ fara ipoteze. $\square$

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

Cuprins

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. $p_1 \land \ldots \land p_k \to q$

  unde $q, p_1, \ldots, p_n$ 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. $p_1 \land \ldots \land p_k \to q$ (o regulă în Prolog `q :- p1,...,pk`)

  unde $q, p_1, \ldots, p_n$ 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. $p_1 \wedge \ldots \wedge p_k \rightarrow q$ (o regulă în Prolog `q :- p1,...,pk`)

  unde $q, p_1, \ldots, p_n$ 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. $p_1 \wedge \ldots \wedge p_k \rightarrow q$ (o regulă în Prolog `q :- p1,...,pk`)

  unde $q, p_1, \ldots, p_n$ sunt variabile propoziționale
- Numim variabilele propoziționale atomi.

Programare logică – cazul logicii propoziționale

- Un "program logic" este o listă $Cd_1, \ldots, Cd_n$ 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. $p_1 \wedge \ldots \wedge p_k \rightarrow q$ (o regulă în Prolog `q :- p1,...,pk`)

  unde $q, p_1, \ldots, p_n$ sunt variabile propoziționale
- Numim variabilele propoziționale atomi.

Programare logică – cazul logicii propoziționale

- Un "program logic" este o listă $Cd_1, \ldots, Cd_n$ de clauze definite.
- O întrebare este o listă $q_1, \ldots, q_m$ 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. $p_1 \wedge \ldots \wedge p_k \to q$ (o regulă în Prolog `q :- p1,...,pk`)

  unde $q, p_1, \ldots, p_n$ sunt variabile propoziționale
- Numim variabilele propoziționale atomi.

Programare logică – cazul logicii propoziționale

- Un "program logic" este o listă $Cd_1, \ldots, Cd_n$ de clauze definite.
- O întrebare este o listă $q_1, \ldots, q_m$ de atomi.
- Sarcina sistemului este să stabilească:

$$Cd_1, \ldots, Cd_n \models q_1 \wedge \ldots \wedge q_m.$$

Clauze propoziționale definite

- O clauză propozițională definită este o formulă care poate avea una din formele:
  1. q (un fapt în Prolog `q.`)
  2. $p_1 \wedge \ldots \wedge p_k \rightarrow q$ (o regulă în Prolog `q :- p1,...,pk`)

  unde $q, p_1, \ldots, p_n$ sunt variabile propoziționale
- Numim variabilele propoziționale atomi.

Programare logică – cazul logicii propoziționale

- Un "program logic" este o listă $Cd_1, \ldots, Cd_n$ de clauze definite.
- O întrebare este o listă $q_1, \ldots, q_m$ de atomi.
- Sarcina sistemului este să stabilească:

$$Cd_1, \ldots, Cd_n \models q_1 \wedge \ldots \wedge q_m.$$

Vom studia metode sintactice pentru a rezolva această problemă!

Sistem de deducție CDP

Sistem de deducție CDP pentru clauze definite propoziționale

Pentru o mulțime $\mathcal{S}$ 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:

$$\frac{P \quad P \to Q}{Q}\ (MP) \qquad\qquad \frac{P \quad Q}{P \land Q}\ (andI)$$

  - Aceste reguli ne permit să deducem formula de sub linie din formulele de deasupra liniei.
  - Sunt regulile $(\to e)$ și $(\land i)$ din deducția naturală pentru logica propozițională.

Sistemul de deducție CDP

$$\frac{P \quad P \to Q}{Q}\ (MP) \qquad\qquad \frac{P \quad Q}{P \land Q}\ (andI)$$

Exemplu

$$\begin{array}{rcl}
\texttt{oslo} & \to & \texttt{windy} \\
\texttt{oslo} & \to & \texttt{norway} \\
\texttt{norway} & \to & \texttt{cold} \\
\texttt{cold} \land \texttt{windy} & \to & \texttt{winterIsComing} \\
 & & \texttt{oslo}
\end{array}$$

Sistemul de deducție CDP

$$\frac{P \quad P \to Q}{Q}\ (MP) \qquad\qquad \frac{P \quad Q}{P \land Q}\ (andI)$$

Exemplu

$$\begin{array}{rcl}
\texttt{oslo} & \to & \texttt{windy} \\
\texttt{oslo} & \to & \texttt{norway} \\
\texttt{norway} & \to & \texttt{cold} \\
\texttt{cold} \land \texttt{windy} & \to & \texttt{winterIsComing} \\
 & & \texttt{oslo}
\end{array}$$

$$\dfrac{\dfrac{\dfrac{oslo \qquad oslo \to norway}{norway} \qquad norway \to cold}{cold} \qquad \dfrac{oslo \qquad oslo \to windy}{windy}}{cold \land windy}$$

Sistemul de deducție CDP

$$\frac{P \quad P \to Q}{Q}\ (MP) \qquad\qquad \frac{P \quad Q}{P \wedge Q}\ (andI)$$

Exemplu

$$\begin{array}{rcl} \texttt{oslo} & \to & \texttt{windy} \\ \texttt{oslo} & \to & \texttt{norway} \\ \texttt{norway} & \to & \texttt{cold} \\ \texttt{cold} \wedge \texttt{windy} & \to & \texttt{winterIsComing} \\ & & \texttt{oslo} \end{array}$$

$$\dfrac{\dfrac{\dfrac{oslo \qquad oslo \to norway}{norway} \qquad norway \to cold}{cold} \qquad \dfrac{oslo \qquad oslo \to windy}{windy}}{cold \wedge windy}$$

$$\frac{cold \wedge windy \qquad cold \wedge windy \to winterIsComing}{winterIsComing}$$

Sistemul de deducție CDP

$$\frac{P \quad P \to Q}{Q}\ (MP) \qquad\qquad \frac{P \quad Q}{P \wedge Q}\ (andI)$$

Exemplu

$$\begin{array}{rcl}
\texttt{oslo} & \to & \texttt{windy} \\
\texttt{oslo} & \to & \texttt{norway} \\
\texttt{norway} & \to & \texttt{cold} \\
\texttt{cold} \wedge \texttt{windy} & \to & \texttt{winterIsComing} \\
 & & \texttt{oslo}
\end{array}$$

1. $oslo \to windy$
2. $oslo \to norway$
3. $norway \to cold$
4. $cold \wedge windy \to winterIsComing$
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 $\mathcal{S}$ în sistemul de deducție CDP, notat

$$\mathcal{S} \vdash Q,$$

dacă există o secvență de formule $Q_1, \ldots, Q_n$ astfel încât $Q_n = Q$ și fiecare Q_i:

- fie aparține lui S
- fie se poate deduce din $Q_1, \ldots, Q_{i-1}$ 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
  - $p_1, \ldots, p_n$ sunt deductibili, și
  - $p_1 \wedge \ldots \wedge p_n \rightarrow r$ 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
  - $p_1, \ldots, p_n$ sunt deductibili, și
  - $p_1 \wedge \ldots \wedge p_n \rightarrow r$ 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
  - $p_1, \ldots, p_n$ sunt deductibili, și
  - $p_1 \wedge \ldots \wedge p_n \rightarrow r$ 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:

$$\frac{P_1, \ldots, P_n \quad P_1 \wedge \cdots \wedge P_n \rightarrow Q}{Q} \ (GMP)$$

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.

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

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

  $$\text{dacă } \mathcal{S} \models q, \text{ atunci } \mathcal{S} \vdash q.$$

  - 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:

$$x_1 \leq x_2 \leq x_3 \leq \ldots$$

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 $x_1 \leq x_2 \leq x_3 \leq \ldots$

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 $x_1 \leq x_2 \leq x_3 \leq \ldots$

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 șir (numărabil) de submulțimi ale lui X $Q_1 \subseteq Q_2 \subseteq \ldots$ evident $\bigcup_n Q_n \in \mathcal{P}(X)$

Funcție monotonă

- Fie $(A, \leq_A)$ și $(B, \leq_B)$ mulțimi parțial ordonate.
  O funcție $f : A \to B$ 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 $f : A \to B$ 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 $f_i : \mathcal{P}(\{1,2,3\}) \to \mathcal{P}(\{1,2,3\})$ 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 $f : A \to B$ 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 $f_i : \mathcal{P}(\{1, 2, 3\}) \to \mathcal{P}(\{1, 2, 3\})$ 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 $f : A \to B$ 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 $f_i : \mathcal{P}(\{1,2,3\}) \to \mathcal{P}(\{1,2,3\})$ 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 $f : A \to B$ 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 $f_i : \mathcal{P}(\{1,2,3\}) \to \mathcal{P}(\{1,2,3\})$ 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 $f : A \to B$ 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 $f_i : \mathcal{P}(\{1,2,3\}) \to \mathcal{P}(\{1,2,3\})$ 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 $f : A \to B$ 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 $f_i : \mathcal{P}(\{1,2,3\}) \to \mathcal{P}(\{1,2,3\})$ 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 $f : A \to B$ 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 $f_i : \mathcal{P}(\{1,2,3\}) \to \mathcal{P}(\{1,2,3\})$ 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 $f : A \to B$ 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 $f : A \to B$ 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 $f : A \to B$ 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 $f_i : \mathcal{P}(\{1,2,3\}) \to \mathcal{P}(\{1,2,3\})$ 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 $f : A \to B$ 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 $f_i : \mathcal{P}(\{1,2,3\}) \to \mathcal{P}(\{1,2,3\})$ 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 $f : A \to B$ 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 $f_i : \mathcal{P}(\{1,2,3\}) \to \mathcal{P}(\{1,2,3\})$ 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 $f : C \to C$ dacă $f(a) = a$.

Teorema de punct fix

- Un element $a \in C$ este punct fix al unei funcții $f : C \to C$ dacă $f(a) = a$.

Teorema Knaster-Tarski pentru CPO

Fie $(C, \leq)$ o mulțime parțial ordonată completă și $\mathbf{F} : C \to C$ o funcție continuă. Atunci

$$a = \bigvee_n \mathbf{F}^n(\bot)$$

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 $f : C \to C$ dacă $f(a) = a$.

Teorema Knaster-Tarski pentru CPO

Fie $(C, \leq)$ o mulțime parțial ordonată completă și $\mathbf{F} : C \to C$ o funcție continuă. Atunci

$$a = \bigvee_n \mathbf{F}^n(\bot)$$

este cel mai mic punct fix al funcției $\mathbf{F}$.

- Observăm că în ipotezele ultimei teoreme secvența

$$\mathbf{F}^0(\bot) = \bot \leq \mathbf{F}(\bot) \leq \mathbf{F}^2(\bot) \leq \cdots \leq \mathbf{F}^n(\bot) \leq \cdots$$

este un lanț, deci $\bigvee_n \mathbf{F}^n(\bot)$ există.

Teorema Knaster-Tarski pentru CPO

Demonstrație

Fie $(C, \leq)$ o mulțime parțial ordonată completă și $\mathbf{F} : C \to C$ 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 $\mathbf{F} : C \to C$ o funcție continuă.

- Arătăm că $a = \bigvee_n \mathbf{F}^n(\perp)$ este punct fix, i.e. $\mathbf{F}(a) = a$

$$\mathbf{F}(a) \quad = \quad \mathbf{F}(\textstyle\bigvee_n \mathbf{F}^n(\perp))$$

Teorema Knaster-Tarski pentru CPO

Demonstrație

Fie $(C, \leq)$ o mulțime parțial ordonată completă și $\mathbf{F} : C \to C$ o funcție continuă.

- Arătăm că $a = \bigvee_n \mathbf{F}^n(\bot)$ este punct fix, i.e. $\mathbf{F}(a) = a$

$$\begin{aligned}
\mathbf{F}(a) &= \mathbf{F}(\textstyle\bigvee_n \mathbf{F}^n(\bot)) \\
&= \textstyle\bigvee_n \mathbf{F}(\mathbf{F}^n(\bot)) \text{ din continuitate}
\end{aligned}$$

Teorema Knaster-Tarski pentru CPO

Demonstrație

Fie $(C, \leq)$ o mulțime parțial ordonată completă și $\mathbf{F} : C \to C$ o funcție continuă.

- Arătăm că $a = \bigvee_n \mathbf{F}^n(\bot)$ este punct fix, i.e. $\mathbf{F}(a) = a$

$$
\begin{aligned}
\mathbf{F}(a) &= \mathbf{F}(\textstyle\bigvee_n \mathbf{F}^n(\bot)) \\
&= \textstyle\bigvee_n \mathbf{F}(\mathbf{F}^n(\bot)) \text{ din continuitate} \\
&= \textstyle\bigvee_n \mathbf{F}^{n+1}(\bot)
\end{aligned}
$$

Teorema Knaster-Tarski pentru CPO

Demonstrație

Fie $(C, \leq)$ o mulțime parțial ordonată completă și $\mathbf{F} : C \to C$ o funcție continuă.

- Arătăm că $a = \bigvee_n \mathbf{F}^n(\bot)$ este punct fix, i.e. $\mathbf{F}(a) = a$

$$
\begin{aligned}
\mathbf{F}(a) &= \mathbf{F}(\textstyle\bigvee_n \mathbf{F}^n(\bot)) \\
&= \textstyle\bigvee_n \mathbf{F}(\mathbf{F}^n(\bot)) \text{ din continuitate} \\
&= \textstyle\bigvee_n \mathbf{F}^{n+1}(\bot) \\
&= \textstyle\bigvee_n \mathbf{F}^n(\bot) = a
\end{aligned}
$$

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 $\mathbf{F}^{n+1}(\perp) \leq b$.

Știm $\mathbf{F}^n(\perp) \leq b$ 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 $\mathbf{F}^{n+1}(\perp) \leq b$.

Știm $\mathbf{F}^n(\perp) \leq b$ 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 At mulțimea variabilelor propozitionale (atomilor) $p_1, p_2, \ldots$ care apar în $\mathcal{S}$.

Clauze definite și funcții monotone

Fie At mulțimea variabilelor propozitionale (atomilor) $p_1, p_2, \ldots$ 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) $p_1, p_2, \ldots$ care apar în $\mathcal{S}$.

Fie $Baza = \{p_i \mid p_i \in \mathcal{S}\}$ mulțimea faptelor din $\mathcal{S}$.

Exemplu

$$\begin{array}{rcl}
\texttt{oslo} & \to & \texttt{windy} \\
\texttt{oslo} & \to & \texttt{norway} \\
\texttt{norway} & \to & \texttt{cold} \\
\texttt{cold} \wedge \texttt{windy} & \to & \texttt{winterIsComing} \\
 & & \texttt{oslo}
\end{array}$$

$At = \{oslo, windy, norway, cold, winterIsComing\}$

$Baza = \{oslo\}$

Clauze definite și funcții monotone

Fie At mulțimea atomilor $p_1, p_2, \ldots$ 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 $f_{\mathcal{S}} : \mathcal{P}(At) \to \mathcal{P}(At)$ prin

$$\boxed{\begin{aligned} f_{\mathcal{S}}(Y) = Y \cup Baza & \\ & \cup \{a \in At \mid (s_1 \wedge \ldots \wedge s_n \to a) \text{ este în } \mathcal{S}, \\ & \qquad s_1 \in Y, \ldots, s_n \in Y\} \end{aligned}}$$

Clauze definite și funcții monotone

Fie At mulțimea atomilor $p_1, p_2, \ldots$ 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 $f_{\mathcal{S}} : \mathcal{P}(At) \to \mathcal{P}(At)$ prin

$$\boxed{\begin{aligned} f_{\mathcal{S}}(Y) = Y \cup Baza & \\ & \cup \{a \in At \mid (s_1 \wedge \ldots \wedge s_n \to a) \text{ este în } \mathcal{S}, \\ & \qquad s_1 \in Y, \ldots, s_n \in Y\} \end{aligned}}$$

Exercițiu. Arătați că funcția $f_{\mathcal{S}}$ este monotonă.

Clauze definite și funcții monotone

Fie At mulțimea atomilor $p_1, p_2, \ldots$ care apar în $\mathcal{S}$.

Propoziție

Funcția $f_{\mathcal{S}} : \mathcal{P}(At) \to \mathcal{P}(At)$ este continuă.

Clauze definite și funcții monotone

Fie At mulțimea atomilor $p_1, p_2, \ldots$ care apar în $\mathcal{S}$.

Propoziție

Funcția $f_{\mathcal{S}} : \mathcal{P}(At) \to \mathcal{P}(At)$ este continuă.

Demonstrație

Arătăm că dacă $Y_1 \subseteq Y_2 \subseteq Y_3 \subseteq \ldots$ atunci $f_{\mathcal{S}}(\bigcup_k Y_k) = \bigcup_k f_{\mathcal{S}}(Y_k)$.

Clauze definite și funcții monotone

Fie At mulțimea atomilor $p_1, p_2, \ldots$ care apar în $\mathcal{S}$.

Propoziție

Funcția $f_{\mathcal{S}} : \mathcal{P}(At) \to \mathcal{P}(At)$ este continuă.

Demonstrație

Arătăm că dacă $Y_1 \subseteq Y_2 \subseteq Y_3 \subseteq \ldots$ atunci $f_{\mathcal{S}}(\bigcup_k Y_k) = \bigcup_k f_{\mathcal{S}}(Y_k)$.

Din faptul că $f_{\mathcal{S}}$ este crescătoare rezultă $f_{\mathcal{S}}(\bigcup_k Y_k) \supseteq \bigcup_k f_{\mathcal{S}}(Y_k)$

Clauze definite și funcții monotone

Fie At mulțimea atomilor $p_1, p_2, \ldots$ care apar în $\mathcal{S}$.

Propoziție

Funcția $f_{\mathcal{S}} : \mathcal{P}(At) \to \mathcal{P}(At)$ este continuă.

Demonstrație

Arătăm că dacă $Y_1 \subseteq Y_2 \subseteq Y_3 \subseteq \ldots$ atunci $f_{\mathcal{S}}(\bigcup_k Y_k) = \bigcup_k f_{\mathcal{S}}(Y_k)$.

Din faptul că $f_{\mathcal{S}}$ este crescătoare rezultă $f_{\mathcal{S}}(\bigcup_k Y_k) \supseteq \bigcup_k f_{\mathcal{S}}(Y_k)$

Demonstrăm în continuare că $f_{\mathcal{S}}(\bigcup_k Y_k) \subseteq \bigcup_k f_{\mathcal{S}}(Y_k)$. Fie $a \in f_{\mathcal{S}}(\bigcup_n Y_k)$. Sunt posibile trei cazuri

Clauze definite și funcții monotone

Fie At mulțimea atomilor $p_1, p_2, \ldots$ care apar în $\mathcal{S}$.

Propoziție

Funcția $f_{\mathcal{S}} : \mathcal{P}(At) \to \mathcal{P}(At)$ este continuă.

Demonstrație

Arătăm că dacă $Y_1 \subseteq Y_2 \subseteq Y_3 \subseteq \ldots$ atunci $f_{\mathcal{S}}(\bigcup_k Y_k) = \bigcup_k f_{\mathcal{S}}(Y_k)$.

Din faptul că $f_{\mathcal{S}}$ este crescătoare rezultă $f_{\mathcal{S}}(\bigcup_k Y_k) \supseteq \bigcup_k f_{\mathcal{S}}(Y_k)$

Demonstrăm în continuare că $f_{\mathcal{S}}(\bigcup_k Y_k) \subseteq \bigcup_k f_{\mathcal{S}}(Y_k)$. Fie $a \in f_{\mathcal{S}}(\bigcup_n Y_k)$. Sunt posibile trei cazuri

- $a \in \bigcup_k Y_k$
  Există un $k \geq 1$ astfel încât $a \in Y_k$, deci $a \in f_{\mathcal{S}}(Y_k) \subseteq \bigcup_k f_{\mathcal{S}}(Y_k)$.

Clauze definite și funcții monotone

Fie At mulțimea atomilor $p_1, p_2, \ldots$ care apar în $\mathcal{S}$.

Propoziție

Funcția $f_{\mathcal{S}} : \mathcal{P}(At) \to \mathcal{P}(At)$ este continuă.

Demonstrație

Arătăm că dacă $Y_1 \subseteq Y_2 \subseteq Y_3 \subseteq \ldots$ atunci $f_{\mathcal{S}}(\bigcup_k Y_k) = \bigcup_k f_{\mathcal{S}}(Y_k)$.

Din faptul că $f_{\mathcal{S}}$ este crescătoare rezultă $f_{\mathcal{S}}(\bigcup_k Y_k) \supseteq \bigcup_k f_{\mathcal{S}}(Y_k)$

Demonstrăm în continuare că $f_{\mathcal{S}}(\bigcup_k Y_k) \subseteq \bigcup_k f_{\mathcal{S}}(Y_k)$. Fie $a \in f_{\mathcal{S}}(\bigcup_n Y_k)$. Sunt posibile trei cazuri

- $a \in \bigcup_k Y_k$
  Există un $k \geq 1$ astfel încât $a \in Y_k$, deci $a \in f_{\mathcal{S}}(Y_k) \subseteq \bigcup_k f_{\mathcal{S}}(Y_k)$.

- $a \in Baza \subseteq \bigcup_k f_{\mathcal{S}}(Y_k)$

Clauze definite și funcții monotone

Fie At mulțimea atomilor $p_1, p_2, \ldots$ care apar în $\mathcal{S}$.

Propoziție

Funcția $f_{\mathcal{S}} : \mathcal{P}(At) \to \mathcal{P}(At)$ este continuă.

Demonstrație

Arătăm că dacă $Y_1 \subseteq Y_2 \subseteq Y_3 \subseteq \ldots$ atunci $f_{\mathcal{S}}(\bigcup_k Y_k) = \bigcup_k f_{\mathcal{S}}(Y_k)$.

Din faptul că $f_{\mathcal{S}}$ este crescătoare rezultă $f_{\mathcal{S}}(\bigcup_k Y_k) \supseteq \bigcup_k f_{\mathcal{S}}(Y_k)$

Demonstrăm în continuare că $f_{\mathcal{S}}(\bigcup_k Y_k) \subseteq \bigcup_k f_{\mathcal{S}}(Y_k)$. Fie $a \in f_{\mathcal{S}}(\bigcup_n Y_k)$. Sunt posibile trei cazuri

- $a \in \bigcup_k Y_k$
  Există un $k \geq 1$ astfel încât $a \in Y_k$, deci $a \in f_{\mathcal{S}}(Y_k) \subseteq \bigcup_k f_{\mathcal{S}}(Y_k)$.
- $a \in Baza \subseteq \bigcup_k f_{\mathcal{S}}(Y_k)$
- Există $s_1, \ldots, s_n$ în $\bigcup_k Y_k$ astfel încât $(s_1 \wedge \ldots \wedge s_n \to a)$ este în $\mathcal{S}$.

Clauze definite și funcții monotone

Demonstrație (cont.)

- Există $s_1, \ldots, s_n$ în $\bigcup_k Y_k$ astfel încât $(s_1 \wedge \ldots \wedge s_n \to a)$ este în $\mathcal{S}$.

Clauze definite și funcții monotone

Demonstrație (cont.)

- Există $s_1, \ldots, s_n$ în $\bigcup_k Y_k$ astfel încât $(s_1 \wedge \ldots \wedge s_n \to a)$ este în $\mathcal{S}$.

  Pentru fiecare $i \in \{1, \ldots, n\}$ există $k_i \in \mathbb{N}$ astfel încât $s_i \in Y_{k_i}$.

Clauze definite și funcții monotone

Demonstrație (cont.)

- Există $s_1, \ldots, s_n$ în $\bigcup_k Y_k$ astfel încât $(s_1 \wedge \ldots \wedge s_n \to a)$ este în $\mathcal{S}$.

  Pentru fiecare $i \in \{1, \ldots, n\}$ există $k_i \in \mathbb{N}$ astfel încât $s_i \in Y_{k_i}$.

  Dacă $k_0 = \max\{k_1, \ldots, k_n\}$ atunci $Y_{k_i} \subseteq Y_{k_0}$ pentru orice $i \in \{1, \ldots, n\}$.

Clauze definite și funcții monotone

Demonstrație (cont.)

- Există $s_1, \ldots, s_n$ în $\bigcup_k Y_k$ astfel încât $(s_1 \wedge \ldots \wedge s_n \to a)$ este în $\mathcal{S}$.

  Pentru fiecare $i \in \{1, \ldots, n\}$ există $k_i \in \mathbb{N}$ astfel încât $s_i \in Y_{k_i}$.

  Dacă $k_0 = \max\{k_1, \ldots, k_n\}$ atunci $Y_{k_i} \subseteq Y_{k_0}$ pentru orice $i \in \{1, \ldots, n\}$.

  Rezultă că $s_1, \ldots, s_n \in Y_{k_0}$, deci $a \in f_{\mathcal{S}}(Y_{k_0}) \subseteq \bigcup_k f_{\mathcal{S}}(Y_k)$.

Demonstrație (cont.)

- Există $s_1, \ldots, s_n$ în $\bigcup_k Y_k$ astfel încât $(s_1 \wedge \ldots \wedge s_n \to a)$ este în $\mathcal{S}$.

  Pentru fiecare $i \in \{1, \ldots, n\}$ există $k_i \in \mathbb{N}$ astfel încât $s_i \in Y_{k_i}$.

  Dacă $k_0 = \max\{k_1, \ldots, k_n\}$ atunci $Y_{k_i} \subseteq Y_{k_0}$ pentru orice $i \in \{1, \ldots, n\}$.

  Rezultă că $s_1, \ldots, s_n \in Y_{k_0}$, 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. $\square$

Clauze definite și funcții monotone

Pentru funcția continuă $f_{\mathcal{S}} : \mathcal{P}(At) \to \mathcal{P}(At)$

$$\boxed{\begin{aligned} f_{\mathcal{S}}(Y) = Y \cup Baza \\ \cup \{a \in At \mid (s_1 \wedge \ldots \wedge s_n \to a) \text{ este în } \mathcal{S}, \\ s_1 \in Y, \ldots, s_n \in Y\} \end{aligned}}$$

aplicând Teorema Knaster-Tarski pentru CPO, obținem că

$$\bigcup_n f_{\mathcal{S}}^n(\emptyset)$$

este cel mai mic punct fix al lui $f_{\mathcal{S}}$.

Clauze definite și funcții monotone

- Analizați ce se întamplă când considerăm succesiv

$$\emptyset, \quad f_{\mathcal{S}}(\emptyset), \quad f_{\mathcal{S}}(f_{\mathcal{S}}(\emptyset)), \quad f_{\mathcal{S}}(f_{\mathcal{S}}(f_{\mathcal{S}}(\emptyset))), \ldots$$

  La fiecare aplicare a lui $f_{\mathcal{S}}$, rezultatul fie se mărește, fie rămâne neschimbat.

Clauze definite și funcții monotone

- Analizați ce se întamplă când considerăm succesiv

$$\emptyset, \quad f_{\mathcal{S}}(\emptyset), \quad f_{\mathcal{S}}(f_{\mathcal{S}}(\emptyset)), \quad f_{\mathcal{S}}(f_{\mathcal{S}}(f_{\mathcal{S}}(\emptyset))), \ldots$$

  La fiecare aplicare a lui $f_{\mathcal{S}}$, rezultatul fie se mărește, fie rămâne neschimbat.

- Să presupunem că în $\mathcal{S}$ avem k atomi. Atunci după $k+1$ aplicări ale lui $f_{\mathcal{S}}$, trebuie să existe un punct în șirul de mulțimi obținute de unde o nouă aplicare a lui $f_{\mathcal{S}}$ nu mai schimbă rezultatul (punct fix):

$$f_{\mathcal{S}}(X) = X$$

Clauze definite și funcții monotone

- Analizați ce se întamplă când considerăm succesiv

$$\emptyset, \quad f_{\mathcal{S}}(\emptyset), \quad f_{\mathcal{S}}(f_{\mathcal{S}}(\emptyset)), \quad f_{\mathcal{S}}(f_{\mathcal{S}}(f_{\mathcal{S}}(\emptyset))), \ldots$$

  La fiecare aplicare a lui $f_{\mathcal{S}}$, rezultatul fie se mărește, fie rămâne neschimbat.

- Să presupunem că în $\mathcal{S}$ avem k atomi. Atunci după $k+1$ aplicări ale lui $f_{\mathcal{S}}$, trebuie să existe un punct în șirul de mulțimi obținute de unde o nouă aplicare a lui $f_{\mathcal{S}}$ nu mai schimbă rezultatul (punct fix):

$$f_{\mathcal{S}}(X) = X$$

- Dacă aplicăm $f_{\mathcal{S}}$ succesiv ca mai devreme până găsim un X cu proprietatea $f_{\mathcal{S}}(X) = X$, atunci găsim cel mai mic punct fix al lui $f_{\mathcal{S}}$.

Cel mai mic punct fix

Exemplu

$$\begin{array}{rcl} cold & \to & wet \\ wet \wedge cold & \to & scotland \end{array}$$

$$\boxed{\begin{array}{l} f_{\mathcal{S}}(Y) = Y \cup Baza \\ \cup\{a \in At \mid (s_1 \wedge \ldots \wedge s_n \to a) \text{ este în } \mathcal{S}, \\ s_1 \in Y, \ldots, s_n \in Y\} \end{array}}$$

Se observă că $f_{\mathcal{S}}(\emptyset) =$

Cel mai mic punct fix

Exemplu

$$\begin{array}{rcl} cold & \to & wet \\ wet \wedge cold & \to & scotland \end{array}$$

$$\boxed{\begin{array}{l} f_{\mathcal{S}}(Y) = Y \cup Baza \\ \cup\{a \in At \mid (s_1 \wedge \ldots \wedge s_n \to a) \text{ este în } \mathcal{S}, \\ s_1 \in Y, \ldots, s_n \in Y\} \end{array}}$$

Se observă că $f_{\mathcal{S}}(\emptyset) = \emptyset$, 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

$$\begin{array}{rcl} cold & & \\ cold & \to & wet \\ windy & \to & dry \\ wet \wedge cold & \to & scotland \end{array}$$

$$\boxed{\begin{array}{l} f_{\mathcal{S}}(Y) = Y \cup Baza \\ \cup\{a \in At \mid (s_1 \wedge \ldots \wedge s_n \to a) \text{ este în } S, \\ s_1 \in Y, \ldots, s_n \in Y\} \end{array}}$$

Exemplu

Exemplu

$$\begin{array}{rcl} cold & & \\ cold & \rightarrow & wet \\ windy & \rightarrow & dry \\ wet \wedge cold & \rightarrow & scotland \end{array}$$

$$\boxed{\begin{array}{l} f_{\mathcal{S}}(Y) = Y \cup Baza \\ \cup\{a \in At \mid (s_1 \wedge \ldots \wedge s_n \rightarrow a) \text{ este în } S, \\ s_1 \in Y, \ldots, s_n \in Y\} \end{array}}$$

$$f_{\mathcal{S}}(\emptyset) \quad = \quad \{\ cold\ \}$$

Exemplu

Exemplu

$$\begin{array}{rcl} cold & & \\ cold & \rightarrow & wet \\ windy & \rightarrow & dry \\ wet \wedge cold & \rightarrow & scotland \end{array}$$

$$\boxed{\begin{array}{l} f_{\mathcal{S}}(Y) = Y \cup Baza \\ \cup\{a \in At \mid (s_1 \wedge \ldots \wedge s_n \rightarrow a) \text{ este în } S, \\ s_1 \in Y, \ldots, s_n \in Y\} \end{array}}$$

$$\begin{array}{rcl} f_{\mathcal{S}}(\emptyset) & = & \{\ cold\ \} \\ f_{\mathcal{S}}(\{\ cold\ \}) & = & \{\ cold, wet\ \} \end{array}$$

Exemplu

Exemplu

$$\begin{array}{rcl} cold & & \\ cold & \to & wet \\ windy & \to & dry \\ wet \wedge cold & \to & scotland \end{array}$$

$$\boxed{\begin{array}{l} f_{\mathcal{S}}(Y) = Y \cup Baza \\ \cup\{a \in At \mid (s_1 \wedge \ldots \wedge s_n \to a) \text{ este în } \mathcal{S}, \\ s_1 \in Y, \ldots, s_n \in Y\} \end{array}}$$

$$\begin{array}{rcl} f_{\mathcal{S}}(\emptyset) & = & \{\ cold\ \} \\ f_{\mathcal{S}}(\{\ cold\ \}) & = & \{\ cold, wet\ \} \\ f_{\mathcal{S}}(\{\ cold, wet\ \}) & = & \{\ cold, wet, scotland\ \} \end{array}$$

Exemplu

Exemplu

$$\begin{array}{rcl} cold & & \\ cold & \rightarrow & wet \\ windy & \rightarrow & dry \\ wet \wedge cold & \rightarrow & scotland \end{array}$$

$$\boxed{\begin{array}{l} f_{\mathcal{S}}(Y) = Y \cup Baza \\ \cup\{a \in At \mid (s_1 \wedge \ldots \wedge s_n \rightarrow a) \text{ este în } S, \\ s_1 \in Y, \ldots, s_n \in Y\} \end{array}}$$

$$\begin{array}{rcl} f_{\mathcal{S}}(\emptyset) & = & \{\ cold\ \} \\ f_{\mathcal{S}}(\{\ cold\ \}) & = & \{\ cold, wet\ \} \\ f_{\mathcal{S}}(\{\ cold, wet\ \}) & = & \{\ cold, wet, scotland\ \} \\ f_{\mathcal{S}}(\{\ cold, wet, scotland\ \}) & = & \{\ cold, wet, scotland\ \} \end{array}$$

Exemplu

Exemplu

$$\begin{array}{rcl} cold & & \\ cold & \rightarrow & wet \\ windy & \rightarrow & dry \\ wet \wedge cold & \rightarrow & scotland \end{array}$$

$$\boxed{\begin{array}{l} f_{\mathcal{S}}(Y) = Y \cup Baza \\ \cup\{a \in At \mid (s_1 \wedge \ldots \wedge s_n \rightarrow a) \text{ este în } S, \\ s_1 \in Y, \ldots, s_n \in Y\} \end{array}}$$

$$\begin{array}{rcl} f_{\mathcal{S}}(\emptyset) & = & \{\ cold\ \} \\ f_{\mathcal{S}}(\{\ cold\ \}) & = & \{\ cold, wet\ \} \\ f_{\mathcal{S}}(\{\ cold, wet\ \}) & = & \{\ cold, wet, scotland\ \} \\ f_{\mathcal{S}}(\{\ cold, wet, scotland\ \}) & = & \{\ cold, wet, scotland\ \} \end{array}$$

Deci cel mai mic punct fix este $\{\ cold, wet, scotland\ \}$.

Programe logice și cel mai mic punct fix

Teoremă

Fie X este cel mai mic punct fix al funcției $f_{\mathcal{S}}$. Atunci

$$q \in X \quad \text{ddacă} \quad \mathcal{S} \models q.$$

Intuiție: Cel mai mic punct fix al funcției $f_{\mathcal{S}}$ este mulțimea tuturor atomilor care sunt consecințe logice ale programului.

Funcția $f_{\mathcal{S}} : \mathcal{P}(At) \to \mathcal{P}(At)$ este definită prin

$$\boxed{\begin{aligned} f_{\mathcal{S}}(Y) = {} & Y \cup Baza \\ & \cup \{a \in At \mid (s_1 \wedge \ldots \wedge s_n \to a) \text{ este în } \mathcal{S}, s_1 \in Y, \ldots, s_n \in Y\} \end{aligned}}$$

unde At este mulțimea atomilor din $\mathcal{S}$ și $Baza = \{p_i \mid p_i \in \mathcal{S}\}$ este mulțimea atomilor care apar în faptele din $\mathcal{S}$.

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 $p_1 \wedge \ldots \wedge p_n \rightarrow r$. 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 $p_1 \land \ldots \land p_n \to r$. Atunci avem două cazuri:
   - există un p_i, $i = 1, \ldots, n$, care nu este în X. Deci $e^+(P) = 1$.

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 $p_1 \wedge \ldots \wedge p_n \rightarrow r$. Atunci avem două cazuri:
   - există un p_i, $i = 1, \ldots, n$, care nu este în X. Deci $e^{+}(P) = 1$.
   - toți p_i, $i = 1, \ldots, n$, sunt în X. Atunci $r \in f_{\mathcal{S}}(X) = X$, deci $e(r) = 1$. În concluzie $e^{+}(P) = 1$.

$\square$

Sistemul de deducție

Corolar

Sistemul de deducție pentru clauze definite propoziționale este complet pentru a arăta clauze unitate:

$$\text{dacă } \mathcal{S} \models q, \text{ atunci } \mathcal{S} \vdash q.$$

Sistemul de deducție

Corolar

Sistemul de deducție pentru clauze definite propoziționale este complet pentru a arăta clauze unitate:

$$\text{dacă } \mathcal{S} \models q, \text{ atunci } \mathcal{S} \vdash q.$$

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.

$\square$

Rezoluție SLD

Metodă de decizie

Avem o metodă de decizie *(decision procedure)* pentru a verifica $\mathcal{S} \vdash q$

Metoda constă în:

- calcularea celui mai mic punct fix X al funcției $f_{\mathcal{S}}$
- dacă $q \in X$ atunci returnăm **true**, altfel returnăm **false**

Metodă de decizie

Avem o metodă de decizie *(decision procedure)* pentru a verifica $\mathcal{S} \vdash q$

Metoda constă în:

- calcularea celui mai mic punct fix X al funcției $f_{\mathcal{S}}$
- 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 $\mathcal{S} \vdash q$

Metoda constă în:

- calcularea celui mai mic punct fix X al funcției $f_{\mathcal{S}}$
- 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 cunoștintelor care pot fi deduse din KB.

Metodă de decizie

Avem o metodă de decizie *(decision procedure)* pentru a verifica $\mathcal{S} \vdash q$

Metoda constă în:

- calcularea celui mai mic punct fix X al funcției $f_{\mathcal{S}}$
- 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 cunoștintelor 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
  - $p_1 \wedge \ldots \wedge p_n \rightarrow q$, unde toate p_i, q sunt variabile propozitionale.
- O clauză definită $p_1 \wedge \ldots \wedge p_n \rightarrow q$ poate fi gândită ca formula

$$\neg p_1 \vee \ldots \vee \neg p_n \vee q$$

Clauze definite

- Singurele formule admise sunt de forma:
  - q
  - $p_1 \wedge \ldots \wedge p_n \rightarrow q$, unde toate p_i, q sunt variabile propozitionale.
- O clauză definită $p_1 \wedge \ldots \wedge p_n \rightarrow q$ poate fi gândită ca formula

$$\neg p_1 \vee \ldots \vee \neg p_n \vee q$$

  Echivalent, putem reprezenta clauza definită de mai sus și prin $\{\neg p_1, \ldots, \neg p_n, q\}$

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:

$\{\neg oslo, windy\}$

$\{\neg oslo, norway\}$

$\{\neg norway, cold\}$

$\{\neg cold, \neg windy, winter\}$

$\{oslo\}$

Calculul celui mai mic punct fix

LFP:

$\{\neg oslo, windy\}$

$\{\neg oslo, norway\}$

$\{\neg norway, cold\}$

$\{\neg cold, \neg windy, winter\}$

$\{oslo\}$

Calculul celui mai mic punct fix

$\{\neg norway, cold\}$

$\{\neg cold, \neg windy, winter\}$

LFP:

$\{oslo\}$

$\{windy\}$

$\{norway\}$

Calculul celui mai mic punct fix

LFP:

$\{\neg norway, cold\}$

$\{\neg cold, \neg windy, winter\}$

$\{oslo\}$

$\{windy\}$

$\{norway\}$

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

$\{\neg \mathit{windy}, \mathit{winter}\}$

LFP:

$\{\mathit{oslo}\}$

$\{\mathit{windy}\}$

$\{\mathit{norway}\}$

$\{\mathit{cold}\}$

Calculul celui mai mic punct fix

$\{\neg windy, winter\}$

LFP:

$\{oslo\}$

$\{windy\}$

$\{norway\}$

$\{cold\}$

Calculul celui mai mic punct fix

LFP:

$\{oslo\}$

$\{windy\}$

$\{norway\}$

$\{cold\}$

$\{\mathbf{winter}\}$

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 satsfiabilității 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.

$$\text{SLD} \quad \boxed{\frac{\neg p_1 \vee \cdots \vee \neg q \vee \cdots \vee \neg p_n}{\neg p_1 \vee \cdots \vee \neg q_1 \vee \cdots \vee \neg q_m \vee \cdots \vee \neg p_n}}$$

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

$$G_0 := \neg q, \quad G_1, \quad \ldots, \quad G_k, \ldots$$

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:

$$\text{KB} = \{\{oslo\}, \{\neg oslo, windy\}, \{\neg oslo, norway\}, \{\neg norway, cold\},$$
$$\{\neg cold, \neg windy, winter\}\}$$

- 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:

  $$\text{KB} = \{\{oslo\}, \{\neg oslo, windy\}, \{\neg oslo, norway\}, \{\neg norway, cold\}, \{\neg cold, \neg windy, winter\}\}$$

- 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 KB $\vdash$ *winter* prin rezoluție SLD:

$\{\neg winter\}$

Exemplu

Demonstrăm KB $\vdash$ *winter* prin rezoluție SLD:

$\{\neg winter\}$ $\qquad\qquad$ $\{\neg cold, \neg windy, winter\}$

$\{\neg cold, \neg windy\}$

Clause Horn propoziționale - rezoluția SLD

Exemplu

Demonstrăm KB $\vdash$ *winter* prin rezoluție SLD:

$\{\neg winter\}$ $\quad$ $\{\neg cold, \neg windy, winter\}$

$\{\neg cold, \neg windy\}$ $\quad$ $\{\neg norway, cold\}$

$\{\neg norway, \neg windy\}$

Clause Horn propoziționale - rezoluția SLD

Exemplu

Demonstrăm KB $\vdash$ *winter* prin rezoluție SLD:

$\{\neg winter\}$ $\quad\{\neg cold, \neg windy, winter\}$

$\{\neg cold, \neg windy\}$ $\quad\{\neg norway, cold\}$

$\{\neg norway, \neg windy\}$ $\quad\{\neg oslo, norway\}$

$\{\neg oslo, \neg windy\}$

Clause Horn propoziționale - rezoluția SLD

Exemplu

Demonstrăm KB $\vdash$ *winter* prin rezoluție SLD:

$\{\neg winter\}$	$\{\neg cold, \neg windy, winter\}$
$\{\neg cold, \neg windy\}$	$\{\neg norway, cold\}$
$\{\neg norway, \neg windy\}$	$\{\neg oslo, norway\}$
$\{\neg oslo, \neg windy\}$	$\{oslo\}$
$\{\neg windy\}$	

Clause Horn propoziționale - rezoluția SLD

Exemplu

Demonstrăm KB $\vdash$ *winter* prin rezoluție SLD:

$\{\neg winter\}$	$\{\neg cold, \neg windy, winter\}$
$\{\neg cold, \neg windy\}$	$\{\neg norway, cold\}$
$\{\neg norway, \neg windy\}$	$\{\neg oslo, norway\}$
$\{\neg oslo, \neg windy\}$	$\{oslo\}$
$\{\neg windy\}$	$\{\neg oslo, windy\}$
$\{\neg oslo\}$	

Clause Horn propoziționale - rezoluția SLD

Exemplu

Demonstrăm KB $\vdash$ *winter* prin rezoluție SLD:

$\{\neg winter\}$	$\{\neg cold, \neg windy, winter\}$
$\{\neg cold, \neg windy\}$	$\{\neg norway, cold\}$
$\{\neg norway, \neg windy\}$	$\{\neg oslo, norway\}$
$\{\neg oslo, \neg windy\}$	$\{oslo\}$
$\{\neg windy\}$	$\{\neg oslo, windy\}$
$\{\neg oslo\}$	$\{oslo\}$
$\square$	

Clause Horn propoziționale - rezoluția SLD

Exemplu

Demonstrăm KB $\vdash$ *winter* prin rezoluție SLD:

$\{\neg winter\}$	$\{\neg cold, \neg windy, winter\}$
$\{\neg cold, \neg windy\}$	$\{\neg norway, cold\}$
$\{\neg norway, \neg windy\}$	$\{\neg oslo, norway\}$
$\{\neg oslo, \neg windy\}$	$\{oslo\}$
$\{\neg windy\}$	$\{\neg oslo, windy\}$
$\{\neg oslo\}$	$\{oslo\}$
$\square$	

În cursurile următoare vom studia aceste mecanisme în logica de ordinul I.

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

Cuprins

Bibliografie:

- Logic Programming, The University of Edinburgh https://www.inf.ed.ac.uk/teaching/courses/lp/
- J.W.Lloyd, Foundations of Logic Programming, 1987

Logica de ordinul I - sintaxa

Limbaj de ordinul I $\mathcal{L}$

- unic determinat de $\tau = (\mathbf{R}, \mathbf{F}, \mathbf{C}, ari)$

Termenii lui $\mathcal{L}$, notați $Trm_{\mathcal{L}}$, 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 $t_1, \ldots, t_n$ sunt termeni, atunci $f(t_1, \ldots, t_n)$ este termen.

Formulele atomice ale lui $\mathcal{L}$ sunt definite astfel:

- dacă $R \in \mathbf{R}$, $ar(R) = n$ și $t_1, \ldots, t_n$ sunt termeni, atunci $R(t_1, \ldots, t_n)$ 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 (opțional)

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 $f^{\mathcal{A}} : A^n \to A$.
- $\mathbf{R}^{\mathcal{A}} = \{R^{\mathcal{A}} \mid R \in \mathbf{R}\}$ 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 $I : V \to A$.

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 $\varphi_1, \ldots, \varphi_n, \varphi$ formule în logica propozițională (enunțuri în calculul cu predicate).

$\{\varphi_1, \ldots, \varphi_n\} \models \varphi$ este echivalent cu

Deducție și satisfiabilitate

Fie $\varphi_1, \ldots, \varphi_n, \varphi$ formule în logica propozițională (enunțuri în calculul cu predicate).

$\{\varphi_1, \ldots, \varphi_n\} \models \varphi$ este echivalent cu

$\models \varphi_1 \wedge \ldots \wedge \varphi_n \rightarrow \varphi$ este echivalent cu

Deducție și satisfiabilitate

Fie $\varphi_1, \ldots, \varphi_n, \varphi$ formule în logica propozițională
(enunțuri în calculul cu predicate).

$\{\varphi_1, \ldots, \varphi_n\} \models \varphi$ este echivalent cu

$\models \varphi_1 \wedge \ldots \wedge \varphi_n \rightarrow \varphi$ este echivalent cu

$\models \neg\varphi_1 \vee \ldots \neg\varphi_n \vee \varphi$ este echivalent cu

Deducție și satisfiabilitate

Fie $\varphi_1, \ldots, \varphi_n, \varphi$ formule în logica propozițională
(enunțuri în calculul cu predicate).

$\{\varphi_1, \ldots, \varphi_n\} \models \varphi$ este echivalent cu

$\models \varphi_1 \wedge \ldots \wedge \varphi_n \to \varphi$ este echivalent cu

$\models \neg\varphi_1 \vee \ldots \neg\varphi_n \vee \varphi$ este echivalent cu

$\varphi_1 \wedge \ldots \wedge \varphi_n \wedge \neg\varphi$ este satisfiabilă

Literali. Clauze

Literali

- În calculul propozițional un literal este o variabilă sau negația unei variabile.

$$\textit{literal} := \; p \mid \neg p \qquad \text{unde } p \text{ este variabilă propozițională}$$

Literali

- În calculul propozițional un literal este o variabilă sau negația unei variabile.

$$literal := \; p \mid \neg p \qquad \text{unde } p \text{ este variabilă propozițională}$$

- În logica de ordinul I un literal este o formulă atomică sau negația unei formule atomice.

$$literal := \; P(t_1, \ldots, t_n) \mid \neg P(t_1, \ldots, t_n)$$

unde $P \in \mathbf{R}, ari(P) = n$, și $t_1, \ldots, t_n$ sunt termeni.

Literali

- În calculul propozițional un literal este o variabilă sau negația unei variabile.

$$\textit{literal} := p \mid \neg p \qquad \text{unde } p \text{ este variabilă propozițională}$$

- În logica de ordinul I un literal este o formulă atomică sau negația unei formule atomice.

$$\textit{literal} := P(t_1, \ldots, t_n) \mid \neg P(t_1, \ldots, t_n)$$

unde $P \in \mathbf{R}, ari(P) = n$, și $t_1, \ldots, t_n$ sunt termeni.

Clauze

- O clauză este o disjuncție de literali.

Clauze

- O clauză este o disjuncție de literali.
- Dacă $L_1, \ldots, L_n$ sunt literali atunci clauza $L_1 \vee \ldots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \ldots, L_n\}$

clauză = mulțime de literali

Clauze

- O clauză este o disjuncție de literali.
- Dacă $L_1, \ldots, L_n$ sunt literali atunci clauza $L_1 \vee \ldots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \ldots, L_n\}$

  clauză = mulțime de literali

- Clauza $C = \{L_1, \ldots, L_n\}$ este satisfiabilă dacă $L_1 \vee \ldots \vee L_n$ este satisfiabilă.

Clauze

- O clauză este o disjuncție de literali.

- Dacă $L_1, \ldots, L_n$ sunt literali atunci clauza $L_1 \vee \ldots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \ldots, L_n\}$

  clauză = mulțime de literali

- Clauza $C = \{L_1, \ldots, L_n\}$ este satisfiabilă dacă $L_1 \vee \ldots \vee L_n$ este satisfiabilă.

- O clauză C este trivială dacă conține un literal și complementul lui.

Clauze

- O clauză este o disjuncție de literali.
- Dacă $L_1, \ldots, L_n$ sunt literali atunci clauza $L_1 \vee \ldots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \ldots, L_n\}$

  clauză = mulțime de literali

- Clauza $C = \{L_1, \ldots, L_n\}$ este satisfiabilă dacă $L_1 \vee \ldots \vee L_n$ 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 $L_1, \ldots, L_n$ sunt literali atunci clauza $L_1 \vee \ldots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \ldots, L_n\}$

  clauză = mulțime de literali

- Clauza $C = \{L_1, \ldots, L_n\}$ este satisfiabilă dacă $L_1 \vee \ldots \vee L_n$ 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 $\{L_1, \dots, L_n\}$

  clauză = 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.

Logica Horn

Clauze în logica de ordinul I

$$\{\neg Q_1, \ldots, \neg Q_n, P_1, \ldots, P_k\}$$

unde $n, k \geq 0$ și $Q_1, \ldots, Q_n, P_1, \ldots, P_k$ sunt formule atomice.

- formula corespunzătoare este

$$\forall x_1 \ldots \forall x_m(\neg Q_1 \vee \ldots \vee \neg Q_n \vee P_1 \vee \ldots \vee P_k)$$

  unde $x_1, \ldots, x_m$ sunt toate variabilele care apar în clauză

- echivalent, putem scrie

$$\forall x_1 \ldots \forall x_m(Q_1 \wedge \ldots \wedge Q_n \rightarrow P_1 \vee \ldots \vee P_k)$$

- cuantificarea universală a clauzelor este implicită

$$Q_1 \wedge \ldots \wedge Q_n \rightarrow P_1 \vee \ldots \vee P_k$$

Clauze definite. Programe logice. Clauze Horn

- clauză:

$$\{\neg Q_1, \ldots, \neg Q_n, P_1, \ldots, P_k\} \quad \text{sau} \quad Q_1 \wedge \ldots \wedge Q_n \rightarrow P_1 \vee \ldots \vee P_k$$

  unde $n, k \geq 0$ și $Q_1, \ldots, Q_n, P_1, \ldots, P_k$ sunt formule atomice.

- clauză program definită: $k = 1$
  - cazul $n > 0$: $Q_1 \wedge \ldots \wedge Q_n \rightarrow P$
  - cazul $n = 0$: $\top \rightarrow P$ (clauză unitate, fapt)

  Program logic definit = mulțime finită de clauze definite

- scop definit (țintă, întrebare): k=0
  - $Q_1 \wedge \ldots \wedge Q_n \rightarrow \bot$

- clauza vidă $\square$: $n = k = 0$

Clauze definite. Programe logice. Clauze Horn

- clauză:

$$\{\neg Q_1, \ldots, \neg Q_n, P_1, \ldots, P_k\} \quad \text{sau} \quad Q_1 \wedge \ldots \wedge Q_n \to P_1 \vee \ldots \vee P_k$$

  unde $n, k \geq 0$ și $Q_1, \ldots, Q_n, P_1, \ldots, P_k$ sunt formule atomice.

- clauză program definită: $k = 1$
  - cazul $n > 0$: $Q_1 \wedge \ldots \wedge Q_n \to P$
  - cazul $n = 0$: $\top \to P$ (clauză unitate, fapt)

  Program logic definit = mulțime finită de clauze definite

- scop definit (țintă, întrebare): k=0
  - $Q_1 \wedge \ldots \wedge Q_n \to \bot$

- clauza vidă $\square$: $n = k = 0$

  Clauza Horn = clauză program definită sau clauză scop ($k \leq 1$)

Clauze Horn țintă

- scop definit (țintă, întrebare): $Q_1 \wedge \ldots \wedge Q_n \to \bot$
  - fie $x_1, \ldots, x_m$ toate variabilele care apar în $Q_1, \ldots, Q_n$

$$\forall x_1 \ldots \forall x_m (\neg Q_1 \vee \ldots \vee \neg Q_n) \mathrel{\text{⫤}\!\!\models} \neg \exists x_1 \ldots \exists x_m (Q_1 \wedge \ldots \wedge Q_n)$$

  - clauza țintă o vom scrie $Q_1, \ldots, Q_n$

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: $P(t_1, \ldots, t_n)$
  - $Q_1 \wedge \ldots \wedge Q_n \rightarrow P$
    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 $Q_1 \wedge \ldots \wedge Q_n$, unde toate Q_i sunt formule atomice

$$KB \models Q_1 \wedge \ldots \wedge Q_n$$

  - Variabilele din KB sunt cuantificate universal.
  - Variabilele din $Q_1, \ldots, Q_n$ sunt cuantificate existențial.

Limbajul PROLOG are la bază logica clauzelor Horn.

Logica clauzelor definite

Exemplu

Fie următoarele clauze definite:

$$
\begin{array}{l}
\mathit{father}(\mathit{jon}, \mathit{ken}).\\
\mathit{father}(\mathit{ken}, \mathit{liz}).\\
\mathit{father}(X, Y) \rightarrow \mathit{ancestor}(X, Y)\\
\mathit{daugther}(X, Y) \rightarrow \mathit{ancestor}(Y, X)\\
\mathit{ancestor}(X, Y) \wedge \mathit{ancestor}(Y, Z) \rightarrow \mathit{ancestor}(X, Z)
\end{array}
$$

Putem întreba:

- $\mathit{ancestor}(\mathit{jon}, \mathit{liz})$
- dacă există Q astfel încât $\mathit{ancestor}(Q, \mathit{ken})$ (adică $\exists Q\, \mathit{ancestor}(Q, \mathit{ken})$)

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*

$$\frac{\theta(Q_1) \quad \theta(Q_2) \quad \ldots \quad \theta(Q_n) \quad (Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n \to P)}{\theta(Q)}$$

unde $Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n \to P \in KB$, iar θ este cgu pentru Q și P.

Sistem de deducție

Exemplu

KB conține următoarele clauze definite:

$father(jon, ken).$
$father(ken, liz).$
$father(X, Y) \to ancestor(X, Y)$
$daugther(X, Y) \to ancestor(Y, X)$
$ancestor(X, Y) \land ancestor(Y, Z) \to ancestor(X, Z)$

$$\frac{\theta(Q_1) \quad \theta(Q_2) \quad \ldots \quad \theta(Q_n) \quad (Q_1 \land Q_2 \land \ldots \land Q_n \to P)}{\theta(Q)}$$

unde $Q_1 \land Q_2 \land \ldots \land Q_n \to P \in KB$, iar θ este cgu pentru Q și P

Sistem de deducție

Pentru o țintă Q, trebuie să găsim o clauză din KB

$$Q_1 \wedge \ldots \wedge Q_n \to P,$$

și un unificator θ pentru Q și P. În continuare vom verifica $\theta(Q_1), \ldots, \theta(Q_n)$.

Sistem de deducție

Pentru o țintă Q, trebuie să găsim o clauză din KB

$$Q_1 \land \ldots \land Q_n \to P,$$

și un unificator θ pentru Q și P. În continuare vom verifica $\theta(Q_1), \ldots, \theta(Q_n)$.

Exemplu

Pentru ținta

$$ancestor(ken, Z),$$

Sistem de deducție

Pentru o țintă Q, trebuie să găsim o clauză din KB

$$Q_1 \wedge \ldots \wedge Q_n \to P,$$

și un unificator θ pentru Q și P. În continuare vom verifica $\theta(Q_1), \ldots, \theta(Q_n)$.

Exemplu

Pentru ținta

$$ancestor(ken, Z),$$

putem folosi o clauză

$$father(Y, X) \to ancestor(Y, X)$$

cu unificatorul

$$\{Y/ken, X/Z\}$$

pentru a obține o nouă țintă

$$father(ken, Z).$$

Sistem de deducție

$$\frac{\theta(Q_1) \quad \theta(Q_2) \quad \ldots \quad \theta(Q_n) \quad (Q_1 \land Q_2 \land \ldots \land Q_n \to P)}{\theta(Q)}$$

unde $Q_1 \land Q_2 \land \ldots \land Q_n \to P \in KB$, iar θ este cgu pentru Q și P.

Exemplu

$$\frac{\dfrac{father(ken, liz)}{father(ken, Z)} \quad (father(Y, X) \to ancestor(Y, X))}{ancestor(ken, Z)}$$

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 țintă.
  - 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 țintă 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 țintă 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 **ȘI**
    - alege noile ținte 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.*

$$KB \models Q \quad \textit{dacă și numai dacă} \quad KB \vdash_b Q$$

Sistemul de inferență *backchain* este corect și complet și pentru formule atomice cu variabile Q:

$$KB \models \exists x Q(x) \text{ dacă și numai dacă } KB \vdash_b \theta(Q)$$
$$\text{pentru o substituție } \theta.$$

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.

$$\text{SLD} \quad \boxed{\dfrac{\neg Q_1 \vee \cdots \vee \neg Q_i \vee \cdots \vee \neg Q_n}{\theta(\neg Q_1 \vee \cdots \vee \neg P_1 \vee \cdots \vee \neg P_m \vee \cdots \vee \neg Q_n)}}$$

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(catelyn).

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

```
?- stark(jonSnow)
```

$$\text{SLD}\quad \boxed{\frac{\neg Q_1 \lor \dots \lor \neg Q_i \lor \dots \lor \neg Q_n}{\theta(\neg Q_1 \lor \dots \lor \neg P_1 \lor \dots \lor \neg P_m \lor \dots \lor \neg Q_n)}}$$

- $Q \lor \neg P_1 \lor \dots \lor \neg P_m$ este o clauză definită din KB
- variabilele din $Q \lor \neg P_1 \lor \dots \lor \neg P_m$ și Q_i se redenumesc
- θ este c.g.u pentru Q_i și Q.

Rezoluția SLD

Exemplu

$father(eddard, sansa)$
$father(eddard, jonSnow)$

$stark(eddard)$
$stark(catelyn)$

$stark(X) \vee \neg father(Y, X) \vee \neg stark(Y)$

$$\frac{\neg stark(jonSnow)}{}$$

$$\theta(X) = jonSnow$$

$$\text{SLD} \quad \boxed{\frac{\neg Q_1 \vee \cdots \vee \neg Q_i \vee \cdots \vee \neg Q_n}{\theta(\neg Q_1 \vee \cdots \vee \neg P_1 \vee \cdots \vee \neg P_m \vee \cdots \vee \neg Q_n)}}$$

- $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ este o clauză definită din KB
- 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

$father(eddard, sansa)$
$father(eddard, jonSnow)$

$stark(eddard)$
$stark(catelyn)$

$stark(X) \vee \neg father(Y, X) \vee \neg stark(Y)$

$$\frac{\neg stark(jonSnow)}{\neg father(Y, jonSnow) \vee \neg stark(Y)}$$

$$\theta(X) = jonSnow$$

$$\text{SLD} \quad \boxed{\frac{\neg Q_1 \vee \cdots \vee \neg Q_i \vee \cdots \vee \neg Q_n}{\theta(\neg Q_1 \vee \cdots \vee \neg P_1 \vee \cdots \vee \neg P_m \vee \cdots \vee \neg Q_n)}}$$

- $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ este o clauză definită din KB
- 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

$father(eddard, sansa)$
$father(eddard, jonSnow)$
$stark(eddard)$
$stark(catelyn)$
$stark(X) \vee \neg father(Y, X) \vee \neg stark(Y)$

$$\frac{\neg stark(jonSnow)}{\neg father(Y, jonSnow) \vee \neg stark(Y)}$$

Rezoluția SLD

Exemplu

$father(eddard, sansa)$
$father(eddard, jonSnow)$
$stark(eddard)$
$stark(catelyn)$
$stark(X) \vee \neg father(Y, X) \vee \neg stark(Y)$

$$\frac{\neg stark(jonSnow)}{\neg father(Y, jonSnow) \vee \neg stark(Y)}$$

$$\frac{\neg father(Y, jonSnow) \vee \neg stark(Y)}{\neg stark(eddard)}$$

Rezoluția SLD

Exemplu

$father(eddard, sansa)$
$father(eddard, jonSnow)$
$stark(eddard)$
$stark(catelyn)$
$stark(X) \vee \neg father(Y, X) \vee \neg stark(Y)$

$$\frac{\neg stark(jonSnow)}{\neg father(Y, jonSnow) \vee \neg stark(Y)}$$

$$\frac{\neg father(Y, jonSnow) \vee \neg stark(Y)}{\neg stark(eddard)}$$

$$\frac{\neg stark(eddard)}{\square}$$

Rezoluția SLD

Fie KB o mulțime de clauze definite și $Q_1 \land \ldots \land Q_m$ o întrebare, unde Q_i sunt formule atomice.

- O derivare din KB prin rezoluție SLD este o secvență

$$G_0 := \neg Q_1 \lor \ldots \lor \neg Q_m, \quad G_1, \quad \ldots, \quad G_k, \ldots$$

  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 $Q_1 \land \ldots \land Q_m$ din KB,*
- $KB \vdash_b Q_1 \land \ldots \land Q_m$,
- $KB \models Q_1 \land \cdots \land Q_m$.

Rezoluția SLD

Teoremă (Completitudinea SLD-rezoluției)

Sunt echivalente:

- *există o SLD-respingere a lui $Q_1 \wedge \ldots \wedge Q_m$ din KB,*
- $KB \vdash_b Q_1 \wedge \ldots \wedge Q_m$,
- $KB \models Q_1 \wedge \cdots \wedge Q_m$.

Demonstrație

Rezultă din completitudinea sistemului de deducție backchain și din faptul că:

există o SLD-respingere a lui $Q_1 \wedge \ldots \wedge Q_m$ din KB

ddacă

$$KB \vdash_b Q_1 \wedge \ldots \wedge Q_m$$

□

Rezoluția SLD - arbori de căutare

Arbori SLD

- Presupunem că avem o mulțime de clauze definite KB și o țintă $G_0 = \neg Q_1 \vee \ldots \vee \neg Q_m$
- 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. $grandfather(X, Z) :- father(X, Y), parent(Y, Z)$
  2. $parent(X, Y) :- father(X, Y)$
  3. $parent(X, Y) :- mother(X, Y)$
  4. $father(ken, diana)$
  5. $mother(diana, brian)$

- Găsiți o respingere din KB pentru

$$:- grandfather(ken, Y)$$

Rezoluția SLD

Exemplu

- Fie KB următoarea mulțime de clauze definite:
  1. $grandfather(X, Z) \vee \neg father(X, Y) \vee \neg parent(Y, Z)$
  2. $parent(X, Y) \vee \neg father(X, Y)$
  3. $parent(X, Y) \vee \neg mother(X, Y)$
  4. $father(ken, diana)$
  5. $mother(diana, brian)$

- Găsiți o respingere din KB pentru

$$\neg grandfather(ken, Y)$$

Rezoluția SLD

Exemplu

1. $grandfather(X, Z) \vee \neg father(X, Y) \vee \neg parent(Y, Z)$
2. $parent(X, Y) \vee \neg father(X, Y)$
3. $parent(X, Y) \vee \neg mother(X, Y)$
4. $father(ken, diana)$
5. $mother(diana, brian)$

$$
\begin{tikzcd}
 & \neg grandfather(ken, Y) \arrow[d, "1"] & \\
 & \neg father(ken, V) \vee \neg parent(V, Y) \arrow[d, "4"] & \\
 & \neg parent(diana, Y) \arrow[dl, "2"'] \arrow[dr, "3"] & \\
\neg father(diana, Y) & & \neg mother(diana, Y) \arrow[d, "5"] \\
 & & \square
\end{tikzcd}
$$

Rezoluția SLD

Exemplu

$$\neg parent(diana, Y)$$

2. $parent(X, Y) \vee \neg father(X, Y)$

$$\xrightarrow{2}$$

$$\neg father(diana, Y)$$

Aplicarea SLD:

Rezoluția SLD

Exemplu

2 $parent(X, Y) \vee \neg father(X, Y)$

$$\neg parent(diana, Y) \xrightarrow{2} \neg father(diana, Y)$$

Aplicarea SLD:

- redenumesc variabilele: $parent(X, Y_2) \vee \neg father(X, Y_2)$
- determin unificatorul: $\theta = X/diana, Y_2/Y$
- aplic regula: $\dfrac{\neg parent(diana, Y)}{\neg father(diana, Y)}$

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).
2.  p(X,X) :- s(X).
3.  q(X,b).
4.  q(b,a).
5.  q(X,a) :- r(a,X).
6.  r(b,a).
7.  s(X) :- t(X,a).
8.  s(X) :- t(X,b).
9.  s(X) :- t(X,X).
10.  t(a,b).
11.  t(b,a).
```

Rezoluția SLD - arbori de căutare

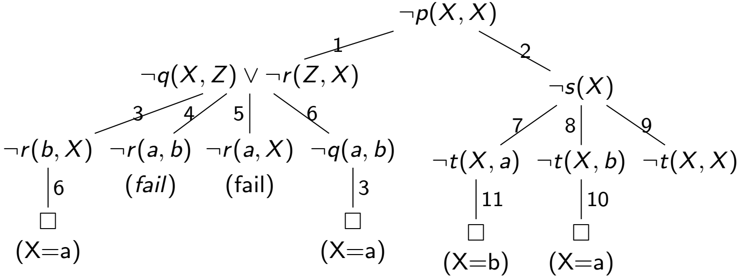
```
1.  p(X,Y) :- q(X,Z), r(Z,Y).
2.  p(X,X) :- s(X).
3.  q(X,b).
4.  q(b,a).
5.  q(X,a) :- r(a,X).
6.  r(b,a).
7.  s(X) :- t(X,a).
8.  s(X) :- t(X,b).
9.  s(X) :- t(X,X).
10. t(a,b).
11. t(b,a).
```

$p(X, Y) \vee \neg q(X, Z) \vee \neg r(Z, Y)$
$p(X, X) \vee \neg s(X)$
$q(X, b)$

$q(b, a)$
$q(X, a) \vee \neg r(a, X)$
$r(b, a)$

$s(X) \vee \neg t(X, a)$
$s(X) \vee \neg t(X, b)$
$s(X) \vee \neg t(X, X)$

$t(a, b)$
$t(b, a)$



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 căutate 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:

$$\begin{array}{rcl}
albedoDecrease & \to & warmerClimate \\
carbonIncrease & \to & warmerClimate \\
warmerClimate & \to & iceMelts \\
iceMelts & \to & albedoDecrease \\
\top & \to & carbonIncrease
\end{array}$$

$$\dfrac{\dfrac{carbonInc. \qquad carbonInc. \to warmerClim.}{warmerClim.} \qquad warmerClim. \to iceMelts}{iceMelts}$$

Pe săptămâna viitoare!

Curs 8-9

2021-2022

Fundamentele limbajelor de programare

Cuprins

Limbajul IMP

Limbajul IMP

Vom implementa un limbaj care conţine:

- Expresii
  - Aritmetice `x + 3`
  - Booleene `x >= 7`
- Instructiuni
  - De atribuire `x = 5`
  - Conditionale `if(x >= 7, x =5, x = 0)`
  - De ciclare `while(x >= 7,x = x - 1)`
- Compunerea instrutiunilor `x=7;while(x>=0,x=x-1)`
- Blocuri de instrucţiuni `{x=7;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

$$
\begin{aligned}
E ::= {} & n \mid x \\
\mid {} & E + E \mid E - E \mid E * E
\end{aligned}
$$

$$
\begin{aligned}
B ::= {} & \texttt{true} \mid \texttt{false} \\
\mid {} & E =< E \mid E >= E \mid E == E \\
\mid {} & \texttt{not}(B) \mid \texttt{and}(B\ ,\ B) \mid \texttt{or}(B\ ,\ B)
\end{aligned}
$$

$$
\begin{aligned}
C ::= {} & \texttt{skip} \\
\mid {} & x = E \\
\mid {} & \texttt{if(}\ B\ ,\ C\ ,\ C\ \texttt{)} \\
\mid {} & \texttt{while(}\ B\ ,\ C\ \texttt{)} \\
\mid {} & \{\ C\ \} \mid C\ ;\ C
\end{aligned}
$$

$$
P ::= \{\ C\ \},\ E
$$

O implementare a limbajului IMP în Prolog

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 semnificatia obisnuită

- pentru valori numerice folosim întregii din Prolog

```
aexp(I) :- integer(I).
```

- pentru identificatori folosim atomii din Prolog

```
aexp(X) :- atom(X).
```

Expresiile aritmetice

$$
\begin{aligned}
E ::= {} & n \mid x \\
& \mid E + E \mid E - E \mid E * E
\end{aligned}
$$

Prolog

```
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

$$
\begin{aligned}
B ::= &\ \texttt{true} \mid \texttt{false}\\
&\mid E \texttt{ =< } E \mid E \texttt{ >= } E \mid E \texttt{ == } E\\
&\mid \texttt{not}(B) \mid \texttt{and}(B\ \texttt{,}\ B) \mid \texttt{or}(B\ \texttt{,}\ B)
\end{aligned}
$$

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 =< 2)).
true.
?- bexp(or(1 =< 2,true)).
true.
?- bexp(or(a =< 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.

Semantica programelor - idei generale

Ce înseamnă semantica formală?

Ce defineşte un limbaj de programare?

Ce înseamnă semantica formală?

Ce defineşte un limbaj de programare?

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

Ce înseamnă semantica formală?

Ce defineşte un limbaj de programare?

- Sintaxa – Simboluri de operaţie, cuvinte cheie, descriere (formală) a programelor/expresiilor bine formate
- Practic – Un limbaj e definit de modul cum poate fi folosit
  - Manual de utilizare şi exemple de bune practici
  - Implementare (compilator/interpretor)
  - Instrumente ajutătoare (analizor de sintaxă, depanator)

Ce înseamnă semantica formală?

Ce defineşte un limbaj de programare?

- Sintaxa – Simboluri de operaţie, cuvinte cheie, descriere (formală) a programelor/expresiilor bine formate
- Practic – Un limbaj e definit de modul cum poate fi folosit
  - Manual de utilizare şi exemple de bune practici
  - Implementare (compilator/interpretor)
  - Instrumente ajutătoare (analizor de sintaxă, depanator)
- Semantica – Ce înseamnă/care e comportamentul unei instrucţiuni?

La ce foloseşte semantica?

- Să înţelegem un limbaj în profunzime
  - Ca programator: pe ce mă pot baza când programez în limbajul dat
  - Ca implementator al limbajului: ce garanţii trebuie să ofer

La ce foloseşte semantica?

- Să înţelegem un limbaj în profunzime
  - Ca programator: pe ce mă pot baza când programez în limbajul dat
  - Ca implementator al limbajului: ce garanţii trebuie să ofer
- Ca instrument în proiectarea unui nou limbaj/a unei extensii
  - Înţelegerea componentelor şi a relaţiilor dintre ele
  - Exprimarea (şi motivarea) deciziilor de proiectare
  - Demonstrarea unor proprietăţi generice ale limbajului

La ce foloseşte semantica?

- Să înţelegem un limbaj în profunzime
  - Ca programator: pe ce mă pot baza când programez în limbajul dat
  - Ca implementator al limbajului: ce garanţii trebuie să ofer
- Ca instrument în proiectarea unui nou limbaj/a unei extensii
  - Înţelegerea componentelor şi a relaţiilor dintre ele
  - Exprimarea (şi motivarea) deciziilor de proiectare
  - Demonstrarea unor proprietăţi 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 instrucţiuni
  - $\vdash \{\varphi\} cod \{\psi\}$
  - modelează un program prin formulele logice pe care le satisface
  - utilă pentru demonstrarea corectitunii

Tipuri de semantică

- Limbaj natural – descriere textuală a efectelor
- Axiomatică – descrierea folosind logică a efectelor unei instrucţiuni
  - $\vdash \{\varphi\} cod \{\psi\}$
  - modelează un program prin formulele logice pe care le satisface
  - utilă pentru demonstrarea corectitunii
- Denotaţională – asocierea unui obiect matematic (denotaţie)
  - $[\![cod]\!]$
  - 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 instrucţiuni
  - $\vdash \{\varphi\} cod \{\psi\}$
  - modelează un program prin formulele logice pe care le satisface
  - utilă pentru demonstrarea corectitunii
- Denotaţională – asocierea unui obiect matematic (denotaţie)
  - $[\![cod]\!]$
  - modelează un program ca obiecte matematice
  - utilă pentru fundamente matematice
- Operaţională – asocierea unei demonstraţii pentru execuţie
  - $\langle cod, \sigma \rangle \to \langle cod', \sigma' \rangle$
  - modelează un program prin execuţia pe o maşină abstractă
  - utilă pentru implementarea de compilatoare şi interpretoare

Tipuri de semantică

- Limbaj natural – descriere textuală a efectelor
- Axiomatică – descrierea folosind logică a efectelor unei instrucţiuni
  - $\vdash \{\varphi\} cod \{\psi\}$
  - modelează un program prin formulele logice pe care le satisface
  - utilă pentru demonstrarea corectitunii
- Denotaţională – asocierea unui obiect matematic (denotaţie)
  - $[\![cod]\!]$
  - modelează un program ca obiecte matematice
  - utilă pentru fundamente matematice
- Operaţională – asocierea unei demonstraţii pentru execuţie
  - $\langle cod, \sigma \rangle \to \langle cod', \sigma' \rangle$
  - modelează un program prin execuţia pe o maşină abstractă
  - utilă pentru implementarea de compilatoare şi interpretoare
- Statică – asocierea unui sistem de tipuri care exclude programe eronate

Semantica small-step

Imagine de ansamblu

- Semantica operatională descrie cum se execută un program pe o masină abstractă (ideală).

Imagine de ansamblu

- Semantica operatională descrie cum se execută un program pe o masină abstractă (ideală).

- Semantica operatională **small-step**
  - semantica structurală, a pasilor mici
  - descrie cum o executie a programului avansează în functie de reduceri succesive.

$$\langle cod, \sigma\rangle \to \langle cod', \sigma'\rangle$$

Imagine de ansamblu

- Semantica operatională descrie cum se execută un program pe o masină abstractă (ideală).
- Semantica operatională **small-step**
  - semantica structurală, a pasilor mici
  - descrie cum o executie a programului avansează în functie de reduceri succesive.

$$\langle cod, \sigma \rangle \rightarrow \langle cod', \sigma' \rangle$$

- Semantica operatională **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 executiei unui program IMP la un moment dat este o funcție parțială (cu domeniu finit):

$$\sigma : Var \rightharpoonup Int$$

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 executiei unui program IMP la un moment dat este o funcție parțială (cu domeniu finit):

$$\sigma : Var \rightharpoonup Int$$

- 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
- Defineste 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 tranziții
  - semantică prin reducere
- Defineste cel mai mic pas de execuție ca o relație „de tranziție” între configurații:

$$\langle cod\ ,\ \sigma \rangle \to \langle cod\ ,\ \sigma' \rangle$$

- Execuția se obține ca o succesiune de astfel de tranziții:

$$\langle x = 0\ ;\ x = x + 1\ ;\ ,\ \perp \rangle \quad \to \quad \langle x = x + 1\ ;\ ,\ x \mapsto 0 \rangle$$

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
  - Semantică Operațională Structurală
  - semantică prin tranziții
  - semantică prin reducere
- Defineste cel mai mic pas de execuție ca o relație „de tranziție” între configurații:

$$\langle cod\ ,\ \sigma\rangle \to \langle cod\ ,\ \sigma'\rangle$$

- Execuția se obține ca o succesiune de astfel de tranziții:

$$\begin{aligned} \langle x = 0\ ;\ x = x + 1\ ;\ ,\ \bot\rangle &\to \langle x = x + 1\ ;\ ,\ x \mapsto 0\rangle \\ &\to \langle x = 0 + 1\ ;\ ,\ x \mapsto 0\rangle \end{aligned}$$

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
  - Semantică Operațională Structurală
  - semantică prin tranziții
  - semantică prin reducere
- Defineste cel mai mic pas de execuție ca o relație „de tranziție” între configurații:

$$\langle cod\ ,\ \sigma\rangle \rightarrow \langle cod\ ,\ \sigma'\rangle$$

- Execuția se obține ca o succesiune de astfel de tranziții:

$$\begin{array}{lcl} \langle x = 0\ ;\ x = x + 1\ ;\ ,\ \bot\rangle & \rightarrow & \langle x = x + 1\ ;\ ,\ x \mapsto 0\rangle \\ & \rightarrow & \langle x = 0 + 1\ ;\ ,\ x \mapsto 0\rangle \\ & \rightarrow & \langle x = 1\ ;\ ,\ x \mapsto 0\rangle \end{array}$$

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
  - Semantică Operațională Structurală
  - semantică prin tranziții
  - semantică prin reducere
- Defineste cel mai mic pas de execuție ca o relație „de tranziție” între configurații:

$$\langle cod\ ,\ \sigma\rangle \to \langle cod\ ,\ \sigma'\rangle$$

- Execuția se obține ca o succesiune de astfel de tranziții:

$$\begin{array}{lll} \langle x = 0\ ;\ x = x + 1\ ;\ ,\ \bot\rangle & \to & \langle x = x + 1\ ;\ ,\ x \mapsto 0\rangle \\ & \to & \langle x = 0 + 1\ ;\ ,\ x \mapsto 0\rangle \\ & \to & \langle x = 1\ ;\ ,\ x \mapsto 0\rangle \\ & \to & \langle \{\}\ ,\ x \mapsto 1\rangle \end{array}$$

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
  - Semantică Operațională Structurală
  - semantică prin tranziții
  - semantică prin reducere
- Defineste cel mai mic pas de execuție ca o relație „de tranziție” între configurații:

$$\langle cod\ ,\ \sigma\rangle \to \langle cod\ ,\ \sigma'\rangle$$

- Execuția se obține ca o succesiune de astfel de tranziții:

$$\begin{aligned}
\langle x = 0\ ;\ x = x + 1\ ;\ ,\ \bot\rangle \quad &\to \quad \langle x = x + 1\ ;\ ,\ x \mapsto 0\rangle \\
&\to \quad \langle x = 0 + 1\ ;\ ,\ x \mapsto 0\rangle \\
&\to \quad \langle x = 1\ ;\ ,\ x \mapsto 0\rangle \\
&\to \quad \langle \{\}\ ,\ x \mapsto 1\rangle
\end{aligned}$$

- Cum definim această relatie?

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
  - Semantică Operațională Structurală
  - semantică prin tranziții
  - semantică prin reducere
- Defineste cel mai mic pas de execuție ca o relație „de tranziție” între configurații:

$$\langle cod\ ,\ \sigma\rangle \to \langle cod\ ,\ \sigma'\rangle$$

- Execuția se obține ca o succesiune de astfel de tranziții:

$$\begin{array}{lll} \langle x = 0\ ;\ x = x + 1\ ;\ ,\ \bot\rangle & \to & \langle x = x + 1\ ;\ ,\ x \mapsto 0\rangle \\ & \to & \langle x = 0 + 1\ ;\ ,\ x \mapsto 0\rangle \\ & \to & \langle x = 1\ ;\ ,\ x \mapsto 0\rangle \\ & \to & \langle \{\}\ ,\ x \mapsto 1\rangle \end{array}$$

- Cum definim această relatie? Prin inductie după elementele din sintaxă.

Redex. Reguli structurale. Axiome

- Expresie reductibilă (redex)
  - Fragmentul de sintaxă care va fi procesat la pasul următor

$$\texttt{if}\,(0 <= 5 + 7 * x\ ,\ r = 1\ ,\ r = 0)$$

Redex. Reguli structurale. Axiome

- Expresie reductibilă (redex)
  - Fragmentul de sintaxă care va fi procesat la pasul următor

$$\texttt{if}\,(0 \mathrel{<=} 5 + 7 * x \ ,\ r = 1 \ ,\ r = 0)$$

Redex. Reguli structurale. Axiome

- Expresie reductibilă (redex)
  - Fragmentul de sintaxă care va fi procesat la pasul următor

$$\texttt{if}\,(0 <= 5 + 7 * x\ ,\ r = 1\ ,\ r = 0)$$

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

$$\texttt{if}\,(0 \mathrel{<=} 5 + 7 * x\ ,\ r = 1\ ,\ r = 0)$$

Reguli structurale

- Folosesc la identificarea următorului redex
- Definite recursiv pe structura termenilor

$$\frac{\langle b\ ,\ \sigma\rangle \to \langle b'\ ,\ \sigma\rangle}{\langle \texttt{if}\,(b, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle \texttt{if}\,(b', bl_1, bl_2)\ ,\ \sigma\rangle}$$

Redex. Reguli structurale. Axiome

- Expresie reductibilă (redex)
  - Fragmentul de sintaxă care va fi procesat la pasul următor

$$\texttt{if}\,(0 <= 5 + 7 * {\color{red}x}\ ,\ r = 1\ ,\ r = 0)$$

Reguli structurale

- Folosesc la identificarea următorului redex
- Definite recursiv pe structura termenilor

$$\frac{\langle b\ ,\ \sigma\rangle \to \langle b'\ ,\ \sigma\rangle}{\langle \texttt{if}\,(b, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle \texttt{if}\,(b', bl_1, bl_2)\ ,\ \sigma\rangle}$$

Axiome

- Realizează pasul computațional

Redex. Reguli structurale. Axiome

- Expresie reductibilă (redex)
  - Fragmentul de sintaxă care va fi procesat la pasul următor

$$\texttt{if}\ (0 <= 5 + 7 * x\ ,\ r = 1\ ,\ r = 0)$$

Reguli structurale

- Folosesc la identificarea următorului redex
- Definite recursiv pe structura termenilor

$$\frac{\langle b\ ,\ \sigma\rangle \rightarrow \langle b'\ ,\ \sigma\rangle}{\langle \texttt{if}\ (b, bl_1, bl_2)\ ,\ \sigma\rangle \rightarrow \langle \texttt{if}\ (b', bl_1, bl_2)\ ,\ \sigma\rangle}$$

Axiome

- Realizează pasul computațional

$$\langle \texttt{if}\ (\texttt{true}, bl_1, bl_2)\ ,\ \sigma\rangle \rightarrow \langle bl_1\ ,\ \sigma\rangle$$

Semantica expresiilor aritmetice

- Semantica unui întreg este o valoare
  - nu poate fi redex, deci nu avem regulă
- Semantica unei variabile

  (ID) $\langle x \,,\, \sigma \rangle \to \langle i \,,\, \sigma \rangle \quad dac\breve{a}\ i = \sigma(x)$
- Semantica adunării a două expresii aritmetice

  (ADD) $\langle i_1 + i_2 \,,\, \sigma \rangle \to \langle i \,,\, \sigma \rangle \quad dac\breve{a}\ i = i_1 + i_2$

  $$\frac{\langle a_1 \,,\, \sigma \rangle \to \langle a_1' \,,\, \sigma \rangle}{\langle a_1 + a_2 \,,\, \sigma \rangle \to \langle a_1' + a_2 \,,\, \sigma \rangle} \qquad \frac{\langle a_2 \,,\, \sigma \rangle \to \langle a_2' \,,\, \sigma \rangle}{\langle a_1 + a_2 \,,\, \sigma \rangle \to \langle a_1 + a_2' \,,\, \sigma \rangle}$$

  Observatie: ordinea de evaluare a argumentelor este nespecificată.

Semantica expresiilor booleene

- Semantica operatorului de comparatie

(LEQ-FALSE) $\langle i_1 \ \texttt{=<} \ i_2 \ , \ \sigma\rangle \rightarrow \langle \texttt{false} \ , \ \sigma\rangle$ *dacă* $i_1 > i_2$

(LEQ-TRUE) $\langle i_1 \ \texttt{=<} \ i_2 \ , \ \sigma\rangle \rightarrow \langle \texttt{true} \ , \ \sigma\rangle$ *dacă* $i_1 \leq i_2$

$$\frac{\langle a_1 \ , \ \sigma\rangle \rightarrow \ \rightarrow\langle a_1' \ , \ \sigma\rangle}{\langle a_1 \ \texttt{=<} \ a_2 \ , \ \sigma\rangle \rightarrow \langle a_1' \ \texttt{=<} \ a_2 \ , \ \sigma\rangle} \qquad \frac{\langle a_2 \ , \ \sigma\rangle \rightarrow \ \rightarrow\langle a_2' \ , \ \sigma\rangle}{\langle a_1 \ \texttt{=<} \ a_2 \ , \ \sigma\rangle \rightarrow \langle a_1 \ \texttt{=<} \ a_2' \ , \ \sigma\rangle}$$

- Semantica negatiei

(!-FALSE) $\langle \texttt{not(true)} \ , \ \sigma\rangle \rightarrow \langle \texttt{false} \ , \ \sigma\rangle$

(!-TRUE) $\langle \texttt{not(false)} \ , \ \sigma\rangle \rightarrow \langle \texttt{true} \ , \ \sigma\rangle$

$$\frac{\langle a \ , \ \sigma\rangle \rightarrow \langle a' \ , \ \sigma\rangle}{\langle \texttt{not} \ (a) \ , \ \sigma\rangle \rightarrow \langle \texttt{not} \ (a') \ , \ \sigma\rangle}$$

Semantica expresiilor booleene

- Semantica operatorului de comparatie
- Semantica negatiei
- Semantica si-ului

(AND-FALSE) $\langle \texttt{and}\,(\texttt{false}, b_2)\,,\,\sigma\rangle \to \langle \texttt{false}\,,\,\sigma\rangle$

(AND-TRUE) $\langle \texttt{and}\,(\texttt{true}, b_2)\,,\,\sigma\rangle \to \langle b_2\,,\,\sigma\rangle$

$$\frac{\langle b_1\,,\,\sigma\rangle \to \langle b_1'\,,\,\sigma\rangle}{\langle \texttt{and}\,(b_1\,,\,b_2)\,,\,\sigma\rangle \to \langle \texttt{and}\,(b_1'\,,\,b_2)\,,\,\sigma\rangle}$$

Semantica compunerii si a blocurilor

- Semantica blocurilor

(Block) $\langle \{\ s\ \}\ ,\ \sigma \rangle \rightarrow \langle s\ ,\ \sigma \rangle$

- Semantica compunerii secventiale

(Next-stmt) $\langle \texttt{skip;}\ s_2\ ,\ \sigma \rangle \rightarrow \langle s_2\ ,\ \sigma \rangle$

$$\frac{\langle s_1\ ,\ \sigma \rangle \rightarrow \langle s_1'\ ,\ \sigma' \rangle}{\langle s_1\ \texttt{;}\ \ s_2\ ,\ \sigma \rangle \rightarrow \rightarrow \langle s_1'\ \texttt{;}\ \ s_2\ ,\ \sigma' \rangle}$$

- Semantica atribuirii

(Asgn) $\langle x = i\ ,\ \sigma \rangle \rightarrow \langle \texttt{skip}\ ,\ \sigma' \rangle$ *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) $\langle \texttt{if}\,(\texttt{true}, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle bl_1\ ,\ \sigma\rangle$

(If-false) $\langle \texttt{if}\,(\texttt{false}, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle bl_2\ ,\ \sigma\rangle$

$$\frac{\langle b\ ,\ \sigma\rangle \to \langle b'\ ,\ \sigma\rangle}{\langle \texttt{if}\,(b, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle \texttt{if}\,(b', bl_1, bl_2)\ ,\ \sigma\rangle}$$

- Semantica lui `while`

(While) $\langle \texttt{while}\,(b, bl)\ ,\ \sigma\rangle \to \langle \texttt{if}\,(b, bl\ ;\texttt{while}\,(b, bl), \texttt{skip})\ ,\ \sigma\rangle$

- Semantica programelor

(Pgm) $$\frac{\langle a_1\ ,\ \sigma_1\rangle \to \langle a_2\ ,\ \sigma_2\rangle}{\langle (\texttt{skip}, a_1)\ ,\ \sigma_1\rangle \to \langle (\texttt{skip}, a_2)\ ,\ \sigma_2\rangle}$$

$$\frac{\langle s_1\ ,\ \sigma_1\rangle \to \langle s_2\ ,\ \sigma_2\rangle}{\langle (s_1, a)\ ,\ \sigma_1\rangle \to \langle (s_2, a)\ ,\ \sigma_2\rangle}$$

Semantica small-step a lui IMP

Execuție pas cu pas

$\langle i = 3 \texttt{ ;while } (0 <= i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } \perp \rangle \xrightarrow{\text{PGM}}$

Semantica small-step a lui IMP

Execuție pas cu pas

$\langle i = 3 \;;\texttt{while}\,(0 <= i \;,\{\; i = i + -4 \;\})\;,\;\perp\rangle \xrightarrow{\textsc{Pgm}}$

$\langle \texttt{while}\,(0 <= i \;,\{\; i = i + -4 \;\})\;,\; i \mapsto 3\rangle \xrightarrow{\textsc{While}}$

Semantica small-step a lui IMP

Execuție pas cu pas

$\langle i = 3\ \texttt{;while}\ (0 <= i\ \texttt{,}\{\ i = i + -4\ \})\ ,\ \perp\rangle \xrightarrow{\textsc{Pgm}}$

$\langle \texttt{while}\ (0 <= i\ \texttt{,}\{\ i = i + -4\ \})\ ,\ i \mapsto 3\rangle \xrightarrow{\textsc{While}}$

$\langle \texttt{if}\ (0 <= i\ ,\ i = i + -4\ \texttt{;while}\ (0 <= i, \{\ i = i + -4\ \})\ \texttt{,skip})\ ,\ i \mapsto 3\rangle \xrightarrow{\textsc{Id}}$

Semantica small-step a lui IMP

Execuție pas cu pas

$$\langle i = 3\ \mathtt{;while}\ (0 \mathrel{<=} i\ \mathtt{,}\{\ i = i + -4\ \})\ ,\ \bot\rangle \xrightarrow{\text{PGM}}$$

$$\langle \mathtt{while}\ (0 \mathrel{<=} i\ \mathtt{,}\{\ i = i + -4\ \})\ ,\ i \mapsto 3\rangle \xrightarrow{\text{WHILE}}$$

$$\langle \mathtt{if}\ (0 \mathrel{<=} i\ \mathtt{,}\ i = i + -4\ \mathtt{;while}\ (0 \mathrel{<=} i, \{\ i = i + -4\ \})\ \mathtt{,skip})\ ,\ i \mapsto 3\rangle \xrightarrow{\text{ID}}$$

$$\langle \mathtt{if}\ (0 \mathrel{<=} 3\ \mathtt{,}\ i = i + -4\ \mathtt{;while}\ (0 \mathrel{<=} i, \{\ i = i + -4\ \})\ \mathtt{,skip})\ ,\ i \mapsto 3\rangle \xrightarrow{\text{LEQ-TRUE}}$$

Semantica small-step a lui IMP

Execuție pas cu pas

$$\langle i = 3\ \texttt{;while}\ (0 \mathrel{<=} i\ \texttt{,\{}\ i = i + -4\ \texttt{\})}\ ,\ \bot\rangle \xrightarrow{\text{PGM}}$$

$$\langle \texttt{while}\ (0 \mathrel{<=} i\ \texttt{,\{}\ i = i + -4\ \texttt{\})}\ ,\ i \mapsto 3\rangle \xrightarrow{\text{WHILE}}$$

$$\langle \texttt{if}\ (0 \mathrel{<=} i\ ,\ i = i + -4\ \texttt{;while}\ (0 \mathrel{<=} i, \{\ i = i + -4\ \})\ \texttt{,skip)}\ ,\ i \mapsto 3\rangle \xrightarrow{\text{ID}}$$

$$\langle \texttt{if}\ (0 \mathrel{<=} 3\ ,\ i = i + -4\ \texttt{;while}\ (0 \mathrel{<=} i, \{\ i = i + -4\ \})\ \texttt{,skip)}\ ,\ i \mapsto 3\rangle \xrightarrow{\text{LEQ-TRUE}}$$

$$\langle \texttt{if}\ (\texttt{true},\ i = i + -4\ \texttt{;while}\ (0 \mathrel{<=} i, \{\ i = i + -4\ \})\ \texttt{,skip)}\ ,\ i \mapsto 3\rangle \xrightarrow{\text{IF-TRUE}}$$

Semantica small-step a lui IMP

Execuție pas cu pas

$\langle i = 3 \;; \texttt{while}\ (0 <= i\ , \{\ i = i + -4\ \})\ ,\ \perp\rangle \xrightarrow{\text{PGM}}$

$\langle \texttt{while}\ (0 <= i\ , \{\ i = i + -4\ \})\ ,\ i \mapsto 3\rangle \xrightarrow{\text{WHILE}}$

$\langle \texttt{if}\ (0 <= i\ ,\ i = i + -4 \;; \texttt{while}\ (0 <= i, \{\ i = i + -4\ \})\ , \texttt{skip})\ ,\ i \mapsto 3\rangle \xrightarrow{\text{ID}}$

$\langle \texttt{if}\ (0 <= 3\ ,\ i = i + -4 \;; \texttt{while}\ (0 <= i, \{\ i = i + -4\ \})\ , \texttt{skip})\ ,\ i \mapsto 3\rangle \xrightarrow{\text{LEQ-TRUE}}$

$\langle \texttt{if}\ (\texttt{true},\ i = i + -4 \;; \texttt{while}\ (0 <= i, \{\ i = i + -4\ \})\ , \texttt{skip})\ ,\ i \mapsto 3\rangle \xrightarrow{\text{IF-TRUE}}$

$\langle i = i + -4 \;; \texttt{while}\ (0 <= i, \{\ i = i + -4\ \})\ ,\ i \mapsto 3\rangle \xrightarrow{\text{ID}}$

$\ldots$

O implementare a semanticii small-step

Semantica small-step

- Defineste cel mai mic pas de executie ca o relatie de tranzitie între 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

  $\langle x \, , \, \sigma \rangle \to \langle i \, , \, \sigma \rangle \quad \textit{dacă } i = \sigma(x)$

Prolog

```
smallstepA(X,S,I,S) :-
  atom(X),
  get(S,X,I).
```

Semantica expresiilor aritmetice

- Semantica adunării a două expresii aritmetice

$\langle i_1 + i_2 \,,\, \sigma \rangle \to \langle i \,,\, \sigma \rangle \quad$ *dacă* $i = i_1 + i_2$

$$\frac{\langle a_1 \,,\, \sigma \rangle \to \langle a_1' \,,\, \sigma \rangle}{\langle a_1 + a_2 \,,\, \sigma \rangle \to \langle a_1' + a_2 \,,\, \sigma \rangle} \qquad \frac{\langle a_2 \,,\, \sigma \rangle \to \langle a_2' \,,\, \sigma \rangle}{\langle a_1 + a_2 \,,\, \sigma \rangle \to \langle a_1 + a_2' \,,\, \sigma \rangle}$$

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

$\langle i_1 =< i_2 \ , \ \sigma \rangle \to \langle \texttt{false} \ , \ \sigma \rangle \quad \textit{dacă } i_1 > i_2$

$\langle i_1 =< i_2 \ , \ \sigma \rangle \to \langle \texttt{true} \ , \ \sigma \rangle \quad \textit{dacă } i_1 \leq i_2$

$$\frac{\langle a_1 \ , \ \sigma \rangle \to \langle a_1' \ , \ \sigma \rangle}{\langle a_1 =< a_2 \ , \ \sigma \rangle \to \langle a_1' =< a_2 \ , \ \sigma \rangle} \qquad \frac{\langle a_2 \ , \ \sigma \rangle \to \langle a_2' \ , \ \sigma \rangle}{\langle a_1 =< a_2 \ , \ \sigma \rangle \to \langle a_1 =< a_2' \ , \ \sigma \rangle}$$

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

$\langle \texttt{not(true)} \ , \ \sigma \rangle \to \langle \texttt{false} \ , \ \sigma \rangle$

$\langle \texttt{not(false)} \ , \ \sigma \rangle \to \langle \texttt{true} \ , \ \sigma \rangle$

$$\frac{\langle a \ , \ \sigma \rangle \to \langle a' \ , \ \sigma \rangle}{\langle \texttt{not} \ (a) \ , \ \sigma \rangle \to \langle \texttt{not} \ (a') \ , \ \sigma \rangle}$$

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

  $\langle \{\ s\ \}\ ,\ \sigma \rangle \to \ \to \langle s\ ,\ \sigma \rangle$

- Semantica compunerii secventiale

  $\langle \{\}\ \ s_2\ ,\ \sigma \rangle \to \langle s_2\ ,\ \sigma \rangle \qquad \dfrac{\langle s_1\ ,\ \sigma \rangle \to \langle s_1'\ ,\ \sigma' \rangle}{\langle s_1\ s_2\ ,\ \sigma \rangle \to \langle s_1'\ s_2\ ,\ \sigma' \rangle}$

Prolog

```
smallstepS({E},S,E,S).

smallstepS((skip;St2),S,St2,S).
smallstepS((St1;St),S1,(St2;St),S2) :- ...
```

Semantica atribuirii

- Semantica atribuirii

  $\langle x = i\ ,\ \sigma\rangle \to \langle \{\}\ ,\ \sigma'\rangle \quad \textit{dacă}\ \sigma' = \sigma[i/x]$

  $$\frac{\langle a\ ,\ \sigma\rangle \to \langle a'\ ,\ \sigma\rangle}{\langle x = a\ ,\ \sigma\rangle \to \langle x = a'\ ;\ ,\ \sigma\rangle}$$

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`

$\langle \texttt{if}\,(\texttt{true}, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle bl_1\ ,\ \sigma\rangle$

$\langle \texttt{if}\,(\texttt{false}, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle bl_2\ ,\ \sigma\rangle$

$$\frac{\langle b\ ,\ \sigma\rangle \to \langle b'\ ,\ \sigma\rangle}{\langle \texttt{if}\,(b, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle \texttt{if}\,(b', bl_1, bl_2)\ ,\ \sigma\rangle}$$

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`

$\langle \texttt{while}\ (b, bl)\ ,\ \sigma\rangle \to \langle \texttt{if}\ (b, bl\ ; \texttt{while}\ (b, bl), \texttt{skip})\ ,\ \sigma\rangle$

Prolog

```
smallstepS(while(BE,St),S,if(BE,(St;while(BE,St)),skip),S).
```

Semantica programelor

- Semantica programelor

$$\frac{\langle a_1 \, , \, \sigma_1 \rangle \rightarrow \langle a_2 \, , \, \sigma_2 \rangle}{\langle (\mathtt{skip}, a_1) \, , \, \sigma_1 \rangle \rightarrow \langle (\mathtt{skip}, a_2) \, , \, \sigma_2 \rangle}$$

$$\frac{\langle s_1 \, , \, \sigma_1 \rangle \rightarrow \langle s_2 \, , \, \sigma_2 \rangle}{\langle (s_1, a) \, , \, \sigma_1 \rangle \rightarrow \langle (s_2, a) \, , \, \sigma_2 \rangle}$$

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 .
```

Pe săptămâna viitoare!

Curs 8-9

2021-2022

Fundamentele limbajelor de programare

Cuprins

Limbajul IMP

Limbajul IMP

Vom implementa un limbaj care conţine:

- Expresii
  - Aritmetice `x + 3`
  - Booleene `x >= 7`
- Instructiuni
  - De atribuire `x = 5`
  - Conditionale `if(x >= 7, x =5, x = 0)`
  - De ciclare `while(x >= 7,x = x - 1)`
- Compunerea instrutiunilor `x=7;while(x>=0,x=x-1)`
- Blocuri de instrucţiuni `{x=7;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

$E ::= n \mid x$
$\quad \mid E + E \mid E - E \mid E * E$

$B ::= \texttt{true} \mid \texttt{false}$
$\quad \mid E =< E \mid E >= E \mid E == E$
$\quad \mid \texttt{not}(B) \mid \texttt{and}(B , B) \mid \texttt{or}(B , B)$

$C ::= \texttt{skip}$
$\quad \mid x = E$
$\quad \mid \texttt{if(} B , C , C \texttt{)}$
$\quad \mid \texttt{while(} B , C \texttt{)}$
$\quad \mid \{ C \} \mid C ; C$

$P ::= \{ C \} , E$

Semantica programelor - idei generale

Ce înseamnă semantica formală?

Ce defineşte un limbaj de programare?

Ce înseamnă semantica formală?

Ce defineşte un limbaj de programare?

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

Ce înseamnă semantica formală?

Ce defineşte un limbaj de programare?

- Sintaxa – Simboluri de operaţie, cuvinte cheie, descriere (formală) a programelor/expresiilor bine formate
- Practic – Un limbaj e definit de modul cum poate fi folosit
  - Manual de utilizare şi exemple de bune practici
  - Implementare (compilator/interpretor)
  - Instrumente ajutătoare (analizor de sintaxă, depanator)

Ce înseamnă semantica formală?

Ce defineşte un limbaj de programare?

- Sintaxa – Simboluri de operaţie, cuvinte cheie, descriere (formală) a programelor/expresiilor bine formate
- Practic – Un limbaj e definit de modul cum poate fi folosit
  - Manual de utilizare şi exemple de bune practici
  - Implementare (compilator/interpretor)
  - Instrumente ajutătoare (analizor de sintaxă, depanator)
- Semantica – Ce înseamnă/care e comportamentul unei instrucţiuni?

La ce foloseşte semantica?

- Să înţelegem un limbaj în profunzime
  - Ca programator: pe ce mă pot baza când programez în limbajul dat
  - Ca implementator al limbajului: ce garanţii trebuie să ofer

La ce foloseşte semantica?

- Să înţelegem un limbaj în profunzime
  - Ca programator: pe ce mă pot baza când programez în limbajul dat
  - Ca implementator al limbajului: ce garanţii trebuie să ofer
- Ca instrument în proiectarea unui nou limbaj/a unei extensii
  - Înţelegerea componentelor şi a relaţiilor dintre ele
  - Exprimarea (şi motivarea) deciziilor de proiectare
  - Demonstrarea unor proprietăţi generice ale limbajului

La ce foloseşte semantica?

- Să înţelegem un limbaj în profunzime
  - Ca programator: pe ce mă pot baza când programez în limbajul dat
  - Ca implementator al limbajului: ce garanţii trebuie să ofer
- Ca instrument în proiectarea unui nou limbaj/a unei extensii
  - Înţelegerea componentelor şi a relaţiilor dintre ele
  - Exprimarea (şi motivarea) deciziilor de proiectare
  - Demonstrarea unor proprietăţi 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 instrucţiuni
  - $\vdash \{\varphi\} cod \{\psi\}$
  - modelează un program prin formulele logice pe care le satisface
  - utilă pentru demonstrarea corectitunii

Tipuri de semantică

- Limbaj natural – descriere textuală a efectelor
- Axiomatică – descrierea folosind logică a efectelor unei instrucţiuni
  - $\vdash \{\varphi\} cod \{\psi\}$
  - modelează un program prin formulele logice pe care le satisface
  - utilă pentru demonstrarea corectitunii
- Denotaţională – asocierea unui obiect matematic (denotaţie)
  - $[\![cod]\!]$
  - 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 instrucţiuni
  - $\vdash \{\varphi\} cod \{\psi\}$
  - modelează un program prin formulele logice pe care le satisface
  - utilă pentru demonstrarea corectitunii
- Denotaţională – asocierea unui obiect matematic (denotaţie)
  - $[\![cod]\!]$
  - modelează un program ca obiecte matematice
  - utilă pentru fundamente matematice
- Operaţională – asocierea unei demonstraţii pentru execuţie
  - $\langle cod, \sigma \rangle \to \langle cod', \sigma' \rangle$
  - modelează un program prin execuţia pe o maşină abstractă
  - utilă pentru implementarea de compilatoare şi interpretoare

Tipuri de semantică

- Limbaj natural – descriere textuală a efectelor
- Axiomatică – descrierea folosind logică a efectelor unei instrucţiuni
  - $\vdash \{\varphi\} cod \{\psi\}$
  - modelează un program prin formulele logice pe care le satisface
  - utilă pentru demonstrarea corectitunii
- Denotaţională – asocierea unui obiect matematic (denotaţie)
  - $[\![cod]\!]$
  - modelează un program ca obiecte matematice
  - utilă pentru fundamente matematice
- Operaţională – asocierea unei demonstraţii pentru execuţie
  - $\langle cod, \sigma \rangle \to \langle cod', \sigma' \rangle$
  - modelează un program prin execuţia pe o maşină abstractă
  - utilă pentru implementarea de compilatoare şi interpretoare
- Statică – asocierea unui sistem de tipuri care exclude programe eronate

Semantica denotaţională

Semantica denotațională

- Introdusă de Christopher Strachey şi Dana Scott (1970)
- Semantica operaţională, ca un interpretor, descrie cum să evaluăm un program.
- Semantica denotaţională, ca un compilator, descrie o traducere a limbajului într-un limbaj diferit cu semantică cunoscută, anume matematica.
- Semantica denotaţională defineşte ce înseamnă un program ca o funcţie matematică.

Semantica denotațională

- Definim stările memoriei ca fiind funcţii parţiale de la mulţimea identificatorilor la mulţimea valorilor:

$$\mathit{State} = \mathit{Id} \rightharpoonup \mathbb{Z}$$

- Asociem fiecărei categorii sintactice o categorie semantică.
- Fiecare construcţie sintactică va avea o denotaţie (interpretare) în categoria semantică respectivă.

Semantica denotațională

- Definim stările memoriei ca fiind funcţii parţiale de la mulţimea identificatorilor la mulţimea valorilor:

$$State = Id \rightharpoonup \mathbb{Z}$$

- Asociem fiecărei categorii sintactice o categorie semantică.
- Fiecare construcţie sintactică va avea o denotaţie (interpretare) în categoria semantică respectivă.De exemplu:
  - denotaţia unei expresii aritmetice este o funcţie parţială de la mulţimea stărilor memoriei la mulţimea valorilor ($\mathbb{Z}$):

$$[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$$

  - denotaţia unei instrucţiuni este o funcţie parţială de la mulţimea stărilor memoriei la mulţimea stărilor memoriei:

$$[[\_]] : Stmt \to (State \rightharpoonup State)$$

Semantica denotațională

$State = Id \rightharpoonup \mathbb{Z}$
$[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
$[[\_]] : Stmt \to (State \rightharpoonup State)$

Atribuirea: $x = \mathrm{expr}$

- Asociem expresiilor aritmetice funcții de la starea memoriei la valori:
- Asociem instrucțiunilor funcții de la starea memoriei la starea (următoare) a memoriei.

Semantica denotațională

$State = Id \rightharpoonup \mathbb{Z}$
$[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
$[[\_]] : Stmt \to (State \rightharpoonup State)$

Atribuirea: $x = \mathrm{expr}$

- 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

$$State = Id \rightharpoonup \mathbb{Z}$$

- Domenii semantice:
  $[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
  $[[\_]] : BExp \to (State \rightharpoonup \{T, F\})$
  $[[\_]] : Stmt \to (State \rightharpoonup State)$

Semantica denotațională: expresii

$$State = Id \rightharpoonup \mathbb{Z}$$

- Domenii semantice:
  $[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
  $[[\_]] : BExp \to (State \rightharpoonup \{T, F\})$
  $[[\_]] : Stmt \to (State \rightharpoonup State)$
- Semantica denotaţională este compoziţională:
  - semantica expresiilor aritmetice
    $[[n]](s) = n$
    $[[x]](s) = s(x)$
    $[[e1 + e2]](s) = [[e1]](s) + [[e2]](s)$

Semantica denotațională: expresii

$$State = Id \rightharpoonup \mathbb{Z}$$

- Domenii semantice:
  $[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
  $[[\_]] : BExp \to (State \rightharpoonup \{T, F\})$
  $[[\_]] : Stmt \to (State \rightharpoonup State)$
- Semantica denotaţională este compoziţională:
  - semantica expresiilor aritmetice
    $[[n]](s) = n$
    $[[x]](s) = s(x)$
    $[[e1 + e2]](s) = [[e1]](s) + [[e2]](s)$
  - semantica expresiilor booleene
    $[[true]](s) = T$, $[[false]](s) = F$
    $[[!b]](s) = \neg b$
    $[[e1 <= e2]](s) = [[e1]](s) <= [[e2]](s)$

Semantica denotaţională: instrucţiuni

$$State = Id \rightharpoonup \mathbb{Z}$$

- Domenii semantice:
  $[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
  $[[\_]] : BExp \to (State \rightharpoonup \{T, F\})$
  $[[\_]] : Stmt \to (State \rightharpoonup State)$

Semantica denotaţională: instrucţiuni

$$State = Id \rightharpoonup \mathbb{Z}$$

- Domenii semantice:
  $[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
  $[[\_]] : BExp \to (State \rightharpoonup \{T, F\})$
  $[[\_]] : Stmt \to (State \rightharpoonup State)$
- Semantica instrucţiunilor:

$$[[\mathtt{skip}]] = id$$

$$[[\mathtt{c1;c2}]] = [[c2]] \circ [[c1]]$$

$$[[\mathtt{x\ =\ e}]](s)(y) = \begin{cases} s(y), \text{ dacă } y \neq x \\ [[e]](s), \text{ dacă } y = x \end{cases}$$

Semantica denotaţională: instrucţiuni

$$State = Id \rightharpoonup \mathbb{Z}$$

- Domenii semantice:
  $[[\_]] : AExp \to (State \rightharpoonup \mathbb{Z})$
  $[[\_]] : BExp \to (State \rightharpoonup \{T, F\})$
  $[[\_]] : Stmt \to (State \rightharpoonup State)$
- Semantica instrucţiunilor:

$$[[\texttt{skip}]] = id$$

$$[[\texttt{c1;c2}]] = [[c2]] \circ [[c1]]$$

$$[[\texttt{x = e}]](s)(y) = \begin{cases} s(y), \text{ dacă } y \neq x \\ [[e]](s), \text{ dacă } y = x \end{cases}$$

$$[[\texttt{if (b) c1 else c2}]](s) = \begin{cases} [[c1]](s), \text{ dacă } [[b]](s) = T \\ [[c2]](s), \text{ dacă } [[b]](s) = F \end{cases}$$

Semantica denotațională

Exemplu

```
if (x<= y) z=x; else z=y;
```

$$[[pgm]](s) = \begin{cases} [[z = x;]](s), \text{ dacă } [[x <= y]](s) = T \\ [[z = y;]](s), \text{ dacă } [[x <= y]](s) = F \end{cases}$$

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 denotaţională pentru `while`?

Mulţimea funcţiilor parţiale

Fie X şi Y două mulţimi.

- $Pfn(X, Y)$ mulţimea funcţiilor parţiale de la X la Y, adică
  $Pfn(X, Y) = X \rightharpoonup Y$
- Pentru $f \in Pfn(X, Y)$ notăm cu $dom(f)$ mulţimea elementelor din X pentru care funcţia este definită.
  Atunci $dom(f) \subseteq X$ şi $f|_{dom(f)} : dom(f) \to Y$ este funcţie.

Mulţimea funcţiilor parţiale

Fie X şi Y două mulţimi.

- $Pfn(X, Y)$ mulţimea funcţiilor parţiale de la X la Y, adică $Pfn(X, Y) = X \rightharpoonup Y$
- Pentru $f \in Pfn(X, Y)$ notăm cu $dom(f)$ mulţimea elementelor din X pentru care funcţia este definită.
  Atunci $dom(f) \subseteq X$ şi $f\,|_{dom(f)} : dom(f) \to Y$ este funcţie.
- Fie $\perp : X \rightharpoonup Y$ unica funcţie cu $dom(\perp) = \emptyset$ (funcţia care nu este definită în nici un punct).
- Definim pe $Pfn(X, Y)$ următoarea relaţie:

$$f \sqsubseteq g \text{ dacă şi numai dacă } dom(f) \subseteq dom(g) \text{ şi } g|_{dom(f)} = f_{dom(f)}$$

Mulţimea funcţiilor parţiale

Fie X şi Y două mulţimi.

- $Pfn(X, Y)$ mulţimea funcţiilor parţiale de la X la Y, adică $Pfn(X, Y) = X \rightharpoonup Y$
- Pentru $f \in Pfn(X, Y)$ notăm cu $dom(f)$ mulţimea elementelor din X pentru care funcţia este definită.
  Atunci $dom(f) \subseteq X$ şi $f|_{dom(f)} : dom(f) \to Y$ este funcţie.
- Fie $\perp : X \rightharpoonup Y$ unica funcţie cu $dom(\perp) = \emptyset$ (funcţia care nu este definită în nici un punct).
- Definim pe $Pfn(X, Y)$ următoarea relaţie:

$$f \sqsubseteq g \text{ dacă şi numai dacă } dom(f) \subseteq dom(g) \text{ şi } g|_{dom(f)} = f_{dom(f)}$$

$(Pfn(X, Y), \sqsubseteq, \perp)$ este CPO

(mulţime parţial ordonată completă în care $\perp$ este cel mai mic element)

$(Pfn(X, Y), \sqsubseteq, \perp)$ este CPO

Exemplu

Definim $\mathbf{F} : Pfn(\mathbb{N}, \mathbb{N}) \to Pfn(\mathbb{N}, \mathbb{N})$ prin

$$\mathbf{F}(g)(k) = \begin{cases} 1, & \text{dacă } k = 0, \\ k * g(k-1) & \text{dacă } k > 0 \text{ şi } (k-1) \in dom(g), \\ \text{nedefinit}, & \text{altfel} \end{cases}$$

- **F** este o funcţie continuă,

$(Pfn(X,Y), \sqsubseteq, \perp)$ este CPO

Exemplu

Definim $\mathbf{F} : Pfn(\mathbb{N},\mathbb{N}) \to Pfn(\mathbb{N},\mathbb{N})$ prin

$$\mathbf{F}(g)(k) = \begin{cases} 1, & \text{dacă } k = 0, \\ k * g(k-1) & \text{dacă } k > 0 \text{ şi } (k-1) \in dom(g), \\ \text{nedefinit}, & \text{altfel} \end{cases}$$

- **F** este o funcţie continuă,deci putem aplica
- Teorema Knaster-Tarski
  Fie $g_n = \mathbf{F}^n(\perp)$ şi $f = \bigvee_n g_n$.
  Ştim că f este cel mai mic punct fix al funcţiei **F**, deci $\mathbf{F}(f) = f$.

$(Pfn(X, Y), \sqsubseteq, \perp)$ este CPO

Exemplu

Definim $\mathbf{F} : Pfn(\mathbb{N}, \mathbb{N}) \to Pfn(\mathbb{N}, \mathbb{N})$ prin

$$\mathbf{F}(g)(k) = \begin{cases} 1, & \text{dacă } k = 0, \\ k * g(k-1) & \text{dacă } k > 0 \text{ şi } (k-1) \in dom(g), \\ \text{nedefinit}, & \text{altfel} \end{cases}$$

- **F** este o funcţie continuă,deci putem aplica

- Teorema Knaster-Tarski
  Fie $g_n = \mathbf{F}^n(\perp)$ şi $f = \bigvee_n g_n$.
  Ştim că f este cel mai mic punct fix al funcţiei **F**, deci $\mathbf{F}(f) = f$.

- Demonstrăm prin inducţie după n că:

  $dom(g_n) = \{0, \ldots, n\}$ şi $g_n(k) = k!$ oricare $k \in dom(g_n)$

- $f : \mathbb{N} \to \mathbb{N}$ este funcţia factorial.

Semantica denotaţională pentru `while`

`while (b) c`

- Definim $\mathbf{F} : \mathit{Pfn}(\mathit{State}, \mathit{State}) \to \mathit{Pfn}(\mathit{State}, \mathit{State})$ prin
- $\mathbf{F}$ este continuă
- Teorema Knaster-Tarski: $\mathit{fix}(\mathbf{F}) = \bigcup_n \mathbf{F}^n(\bot)$

Semantica denotaţională pentru `while`

`while (b) c`

- Definim $\mathbf{F} : Pfn(State, State) \to Pfn(State, State)$ prin

$$\mathbf{F}(g)(s) = \begin{cases} g([[c]](s)) & \text{dacă } [[b]](s) = T \\ s & \text{dacă } [[b]](s) = F \\ \text{nedefinit,} & \text{altfel} \end{cases}$$

- **F** este continuă
- Teorema Knaster-Tarski: $fix(\mathbf{F}) = \bigcup_n \mathbf{F}^n(\bot)$

Semantica denotaţională pentru `while`

`while (b) c`

- Definim $\mathbf{F} : \mathit{Pfn}(\mathit{State}, \mathit{State}) \to \mathit{Pfn}(\mathit{State}, \mathit{State})$ prin

$$\mathbf{F}(g)(s) = \begin{cases} g([[c]](s)) & \text{dacă } [[b]](s) = T \\ s & \text{dacă } [[b]](s) = F \\ \text{nedefinit,} & \text{altfel} \end{cases}$$

- $\mathbf{F}$ este continuă
- Teorema Knaster-Tarski: $\mathit{fix}(\mathbf{F}) = \bigcup_n \mathbf{F}^n(\perp)$
- Semantica denotaţională:

$$[[\_]] : \mathit{Stmt} \to (\mathit{State} \rightharpoonup \mathit{State})$$

$$[[\texttt{while (b) c}]](s) = \mathit{fix}(\mathbf{F})(s)$$

Avantaje şi dezavantaje

Semantica operațională

- + Defineşte precis noțiunea de pas computațional
- + Semnalează erorile, oprind execuția
- + Execuția devine uşor de urmărit şi depanat
- – Regulile structurale sunt evidente şi deci plictisitor de scris
- – Nemodular: adăugarea unei trăsături noi poate solicita schimbarea întregii definiții

Semantica denotațională

- + Formală, matematică, foarte precisă
- + Compoziţională (morfisme şi compuneri de funcţii)
- – Domeniile devin din ce în ce mai complexe.

Semantica axiomatică

Semantica Axiomatică

- Inventată de 1969 Tony Hoare în 1969 (insiprată de rezultatele lui Robert Floyd).
- Defineşte triplete (triplete Hoare) de forma

$$\{Pre\}\ S\ \{Post\}$$

unde:

  - *S* este o instrucțiune (Stmt)
  - *Pre* (precondiție), respectiv *Post* (postcondiție) sunt aserțiuni logice asupra stării sistemului înaintea, respectiv după execuția lui *S*
  - Limbajul aserţiunilor este un limbaj de ordinul I.

- Tripletul $\{Pre\}\ S\ \{Post\}$ este (parţial) corect dacă:
  - dacă programul se execută dintr-o stare iniţială care satisface *Pre*
  - şi execuţia se termină
  - atunci se ajunge într-o stare finală care satisface *Post*.

Semantica Axiomatică

Defineşte triplete (triplete Hoare) de forma

$$\{Pre\}\ S\ \{Post\}$$

- Tripletul $\{Pre\}\ S\ \{Post\}$ este (parţial) corect dacă:
  - dacă programul se execută dintr-o stare iniţială care satisface *Pre*
  - şi execuţia se termină
  - atunci se ajunge într-o stare finală care satisface *Post*.

Exemplu

- $\{\mathtt{x} = 1\}$ x = x+1 $\{\mathtt{x} = 2\}$ este corect
- $\{\mathtt{x} = 1\}$ `x = x+1` $\{\mathtt{x} = 3\}$ **nu** este corect
- $\{\top\}$ `if (x<=y) z=x; else z=y;` $\{\mathtt{z} = \mathit{min}(\mathtt{x}, \mathtt{y})\}$ este corect

Semantica Axiomatică

Defineşte triplete (triplete Hoare) de forma

$$\{Pre\}\ S\ \{Post\}$$

unde:

- S este o instrucțiune (Stmt)
- Pre (precondiție), respectiv $Post$ (postcondiție) sunt aserțiuni logice asupra stării sistemului înaintea, respectiv după execuția lui S

Se asociază fiecărei construcții sintactice Stmt o regulă de deducție care defineşte recursiv tripletele Hoare descrise mai sus.

Sistem de reguli pentru logica Floyd-Hoare

$$(\rightarrow) \quad \frac{P1 \rightarrow P2 \quad \{P2\}\ c\ \{Q2\} \quad Q2 \rightarrow Q1}{\{P1\}\ c\ \{P2\}}$$

$$(\vee) \quad \frac{\{P1\}\ c\ \{Q\} \quad \{P2\}\ c\ \{Q\}}{\{P1 \vee P2\}\ c\ \{Q\}}$$

$$(\wedge) \quad \frac{\{P\}\ c\ \{Q1\} \quad \{P\}\ c\ \{Q2\}}{\{P\}\ c\ \{Q1 \wedge Q2\}}$$

Logica Floyd-Hoare pentru IMP1

$$(\textsc{Skip}) \quad \frac{\cdot}{\{P\}\ \{\}\ \{P\}}$$

$$(\textsc{Seq}) \quad \frac{\{P\}\ c1\ \{Q\} \quad \{Q\}\ c2\ \{R\}}{\{P\}\ c1; c2\ \{R\}}$$

$$(\textsc{Asign}) \quad \frac{}{\{P[x/e]\}\ x = e;\ \{P\}}$$

$$(\textsc{If}) \quad \frac{\{b \wedge P\}\ c1\ \{Q\} \quad \{\neg b \wedge P\}\ c2\ \{Q\}}{\{P\}\ \texttt{if}\ (b) c1\ \texttt{else}\ c2\ \{Q\}}$$

$$(\textsc{While}) \quad \frac{\{b \wedge P\}\ c\ \{P\}}{\{P\}\ \texttt{while}\ (b)\ c\ \{\neg b \wedge P\}}$$

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

$\{\mathtt{x} + \mathtt{y} = \mathtt{y} + 10\}$ `x = x+ y` $\{\mathtt{x} = \mathtt{y} + 10\}$

- regula pentru condiţii

$$(\text{IF}) \quad \frac{\{b \wedge P\}\ c1\ \{Q\} \quad \{\neg b \wedge P\}\ c2\ \{Q\}}{\{P\}\ \mathtt{if}\ (b) c1\ \mathtt{else}\ c2\ \{Q\}}$$

Exemplu

Pentru a demonstra $\{\top\}$ `if (x<=y) z=x; else z=y;` $\{\mathtt{z} = min(\mathtt{x}, \mathtt{y})\}$
este suficient să demonstrăm $\{\mathtt{x} \leq \mathtt{y}\}$ `z=x;` $\{\mathtt{z} = min(\mathtt{x}, \mathtt{y})\}$
şi $\{\neg(\mathtt{x} \leq \mathtt{y})\}$ `z=y;` $\{\mathtt{z} = min(\mathtt{x}, \mathtt{y})\}$

Invarianţi pentru `while`

Cum demonstrăm $\{P\}$ `while (b)c` $\{Q\}$?

- Se determină un invariant I şi se foloseşte următoarea regulă:

$$(\textsc{Inv})\quad \frac{P \to I \quad \{b \land I\}\ c\ \{I\} \quad (I \land \neg b) \to Q}{\{P\}\ \texttt{while}\ (b)\ c\ \{Q\}}$$

Invarianţi pentru `while`

Cum demonstrăm $\{P\}$ `while (b)c` $\{Q\}$?

- Se determină un invariant I şi se foloseşte următoarea regulă:

$$(\textsc{Inv})\quad \frac{P \to I \quad \{b \wedge I\}\ c\ \{I\} \quad (I \wedge \neg b) \to Q}{\{P\}\ \texttt{while}\ (b)\ c\ \{Q\}}$$

Invariantul trebuie să satisfacăurmătoarele proprietăţi:

- să fie adevărat iniţial
- să rămână adevărat după executarea unui ciclu
- să implice postcondiţia la ieşirea din buclă

Invarianţi pentru `while`

$\{\mathtt{x} = 0 \ \wedge \ 0 \leq n \ \wedge \mathtt{y} = 1\}$

```
while (x < n) { x = x + 1; y = y * x;}
```

$\{\mathtt{y} = n!\}$

Invarianţi pentru `while`

$\{\mathtt{x} = 0 \ \land\ 0 \leq n \ \land \mathtt{y} = 1\}$

```
while (x < n) { x = x + 1; y = y * x;}
```

$\{\mathtt{y} = n!\}$

- Invariantul I este $\mathtt{y} = x!$

Invarianţi pentru `while`

$\{\mathtt{x} = 0 \ \wedge \ 0 \leq n \ \wedge \mathtt{y} = 1\}$

```
while (x < n) { x = x + 1; y = y * x;}
```

$\{\mathtt{y} = n!\}$

- Invariantul I este $\mathtt{y} = x!$
- $(\mathtt{x} = 0 \ \wedge \ 0 \leq n \ \wedge \mathtt{y} = 1) \to I$
- $\{I \wedge (x < n)\}$ `x = x + 1; y = y * x;` $\{I\}$
- $I \wedge \neg(\mathtt{x} < n) \to (\mathtt{y} = n!)$

Dafny

- Dezvoltat la Microsoft Research
- Un limbaj imperativ compilat open-source
- Suportă demonstraţii formale folosind precondiţii, postcondiţii, invarianţi de bucle
- Demonstrează şi terminarea programelor
- Concepte din diferite paradigme de programare
  - Programare imperativă: `if`, `while`, `:=`, ...
  - Programare funcţională: `function`, `datatype`, ...
  - ...

Dafny

- Pagina limbajului Dafny

  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 postcondiţii!

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 postcondiţii!

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 condiţia de mai sus?

Dafny - maximul a două numere

Adăugăm postcondiţii!

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 condiţia de mai sus?
O metodă este reprezentată prin precondiţiile şi postcondiţiile pe care le satisface, iar codul este "ignorat" ulterior.
În exemplul de mai sus, pentru `x = 3` şi `y = 6`, `z = 7` satisface postcondiţia (dacă ignorăm codul) dar nu este maximul dintre `x` şi `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 postcondiţia!

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 reuşeşte să o demonstreze!

Dafny - suma primelor n numere impare

Adăugăm postcondiţia!

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 reuşeşte să o demonstreze! Trebuie să ii spunem ce se întâmplă în buclă prin invarianţi.

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 reuşeşte să demonstreze terminarea!
Adăugăm un nou invariant care ne asigură că *i* nu depaşeşte *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);
 }
}
```

Totuşi Dafny nu reuşeşte să demonstreze postcondiţia! 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);
 }
}
```

Totuşi Dafny nu reuşeşte să demonstreze postcondiţia! 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 - funcţia 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;
    }}
```

Pe săptămâna viitoare!

Curs 11

Implementarea Mini-Haskell în Haskell

Mini-Haskell

Vom defini folosind Haskell un mini limbaj funcţional şi semantica lui denotaţională.

- Limbajul Mini-Haskell conţine:
  - expresii de tip **Int**
  - expresii de tip funcţie (λ-expresii)
  - expresii provenite din aplicarea functiilor
- Pentru a defini semantica limbajului vom introduce domeniile semantice (valorile) asociate expresiilor limbajului.
- Pentru a evalua (interpreta) expresiile vom defini un mediu de evaluare in care vom retine variabilele si valorile curente asociate.

Mini-Haskell (λ-calcul cu întregi). Sintaxă

```haskell
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)
```

Domenii

Domeniul valorilor

```
data Value = Num Integer
           | Fun (Value -> Value)
           | Wrong -- pentru reprezentarea erorilor
```

Mediul de evaluare

```
type Environment = [(Name, Value)]
```

Domeniul de evaluare

Fiecărei 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
```

Evaluarea expresiilor Mini-Haskell în Haskell

```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

```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 -> String
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 -> String
testIO t =  putStrLn . show $ interp t []

> test pgm
42
```

Ce este **IO**?

Curs 11

Cuprins

Monade

Haskell

- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții.
- Funcțiile sunt pure: aceleași rezultate pentru aceleași intrări.
- Puritatea asigură consistență:
  - O bucată de cod nu poate corupe datele altei bucăți 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: aceleași rezultate pentru aceleași intrări.

- Puritatea asigură consistență:
  - O bucată de cod nu poate corupe datele altei bucăți de cod.
  - Mai ușor de testat decât codul care interacționează cu mediul.

Cum interacționezi cu mediul extern, păstrând puritatea?

Haskell

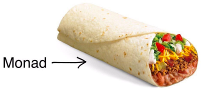
- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții.
- Funcțiile sunt pure: aceleași rezultate pentru aceleași intrări.
- Puritatea asigură consistență:
  - O bucată de cod nu poate corupe datele altei bucăți de cod.
  - Mai ușor de testat decât codul care interacționează cu mediul.

Cum interacționezi cu mediul extern, păstrând puritatea?
Se folosesc *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/

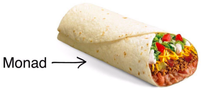


https://twitter.com/monadburritos

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/



https://twitter.com/monadburritos

- "All told, a monad in X is just a monoid in the category of endofunctors in X, with product x replaced by composition of endofunctors and unit set by the identity endofunctor."
  Saunders Mac Lane, Categories for the Working Mathematician, 1998.

Funcţii îmbogăţite şi efecte

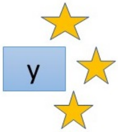
- Funcţie simplă: $x \mapsto y$

  ştiind x, obţinem direct y

Funcţii îmbogăţite şi efecte

- Funcţie simplă: $x \mapsto y$

  ştiind x, obţinem direct y

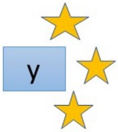
- Funcţie îmbogăţită: $x \mapsto$ 

  ştiind x, putem să extragem y şi producem un efect

Funcţii îmbogăţite şi efecte

- Funcţie simplă: $x \mapsto y$

  ştiind x, obţinem direct y

- Funcţie îmbogăţită: $x \mapsto$ 

  ştiind x, putem să extragem y şi producem un efect

Referinţe:
https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/

https://bartoszmilewski.com/2016/11/30/monads-and-effects/

Funcţii îmbogăţite şi efecte

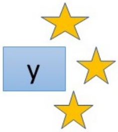
Funcţie îmbogăţită: $x \mapsto$

Exemplu

- Folosind tipul **Maybe** a

```haskell
data Maybe a = Nothing | Just a
f :: Int -> Maybe Int
f  x = if x < 0 then Nothing else (Just x)
```

Funcţii îmbogăţite şi efecte

Funcţie îmbogăţită: $x \mapsto$ 

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")

Compunerea funcţiilor

- Principala operaţie pe care o facem cu funcţii este compunerea

```
f :: a -> b , g :: b -> c , g . f :: a -> c
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Valoarea de tip b este transmisă direct funcţiei g.

Compunerea funcţiilor

- Principala operaţie pe care o facem cu funcţii este compunerea

```
f :: a -> b , g :: b -> c , g . f :: a -> c
(.) :: (b -> c) -> (a -> b) -> a -> c
```

  Valoarea de tip b este transmisă direct funcţiei g.

- Ce facem dacă

```
f :: a -> m b , g :: b -> m c
```

  unde m este un constructor de tip care îmbogăţeşte tipul?

Compunerea funcţiilor

- Principala operaţie pe care o facem cu funcţii este compunerea

```
f :: a -> b , g :: b -> c , g . f :: a -> c
(.) :: (b -> c) -> (a -> b) -> a -> c
```

  Valoarea de tip b este transmisă direct funcţiei g.

- Ce facem dacă

```
f :: a -> m b , g :: b -> m c
```

  unde m este un constructor de tip care îmbogăţeşte tipul?

  De exemplu,
  - m = **Maybe**
  - m = Writer **log**

- Atenţie! m trebuie să aibă un singur argument.

Compunerea funcţiilor

`f :: a -> m b , g :: b -> m c`

unde m este un constructor de tip care îmbogăţeşte tipul.

Vrem să definim o "compunere" pentru funcţii îmbogăţite

(<=<) :: (b -> m c) -> (a -> m b) -> a -> m c

Atunci când definim g <=< f trebuie să extragem valoarea întoarsă de f şi să o trimitem lui g.

Exemplu: logging în Haskell

„Îmbogățim” rezultatul funcției cu mesajul de log.

```
newtype Writer log a = Writer { runWriter :: (a, log)
    }

logIncrement :: Int -> Writer String Int
logIncrement x = Writer
    (x + 1, "Called increment with argument " ++ show
        x ++ "\n")
```

Problemă: Cum calculăm logIncrement (logIncrement x)?

Exemplu: logging în Haskell

```
newtype Writer log a = Writer { runWriter :: (a, log)
    }

logIncrement :: Int -> Writer String Int
logIncrement x = Writer
    (x + 1, "Called increment with argument " ++ show
        x ++ "\n")


logIncrement2 :: Int -> Writer String Int
logIncrement2 x = let (y, log1) = runWriter (
    logIncrement x)
                      (z, log2) = runWriter (
                          logIncrement y)
                  in  Writer (z, log1 ++ log2)
```

Cum compunem funcții cu efecte laterale

Problema generală

Dată fiind funcția f :: a -> m b și funcția g :: b -> m c, vreau să obțin o funcție g <=< f :: a -> m c care este „compunerea” lui g și f, propagând efectele laterale.

Exemplu

```
> logIncrement x = Writer (x + 1, "Called increment
    with argument " ++ show x ++ "\n")

> logIncrement <=< logIncrement $ 3
Writer {runWriter = (5,"Called increment with
    argument 3\nCalled increment with argument 4\n")}
```

Observaţie: Funcţia (<=<) 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 >> mb = ma >>= \_ -> mb
```

- m a este tipul computațiilor care produc rezultate de tip a (și au efecte laterale)
- a –> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a computațiilor

Clasa de tipuri **Monad**

```haskell
class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a - > m b -> m b
    return :: a -> m a

    ma >> mb = ma >>= \_ -> mb
```

- m a este tipul computațiilor care produc rezultate de tip a (și au efecte laterale)
- a -> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a computațiilor
- în Control.**Monad** sunt definite
  - f >=>g = \x -> f x >>= g
  - (<=<) = **flip** (>=>)

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 >> mb = ma >>= \_ -> mb
```

- m a este tipul computațiilor care produc rezultate de tip a (și au efecte laterale)
- a -> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a computațiilor
- în Control.**Monad** sunt definite
  - f >=>g = \x -> f x >>= g
  - (<=<) = **flip** (>=>)

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 >> mb = ma >>= \_ -> mb
```

- m a este tipul computațiilor care produc rezultate de tip a (și au efecte laterale)
- a –> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= 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 şi element neutru

Operația <=< de compunere a funcţiilor îmbogăţite este asociativă şi are element neutru **return**

Proprietățile monadelor

Asociativitate şi element neutru

Operația <=< de compunere a funcţiilor îmbogăţite este asociativă şi are element neutru **return**

- Element neutru (la dreapta): g <=< **return** = g

```
(return x) >>= g = g x
```

- Element neutru (la stânga): **return** <=< g = g

```
x >>= return = x
```

- Associativitate: h <=< (g <=< f) = (h <=< g) <=< f

```
(f >>= g) >>= h = f >>= (\ x -> (g x >>= h))
```

Notația **do** pentru monade

Notaţia cu operatori	Notaţia **do**
e >>= \x -> rest	x <- e rest
e >>= _ -> rest	e rest
e >> rest	e rest

Notația **do** pentru monade

Notaţia cu operatori	Notaţia **do**
e >>= \x –> rest	x <– e rest
e >>= _ –> rest	e rest
e >> rest	e rest

De exemplu

```
e1    >>=  \x1  ->
e2    >>  e3
```

devine

Notația **do** pentru monade

Notaţia cu operatori	Notaţia **do**
e >>= \x −> rest	x <− e rest
e >>= _ −> rest	e rest
e >> 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**
Parțialitate	Monada **Maybe**
Nedeterminism	Monada [] (listă)
Memorie read-only	Monada Reader

Monada **Maybe**(a funcțiilor parțiale)

```haskell
data Maybe a = Nothing | Just a
```

Monada **Maybe**(a funcțiilor parțiale)

```haskell
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)

```haskell
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 excepțiilor)

```
data Either err a = Left err | Right a
```

Monada **Either**(a excepțiilor)

```haskell
data Either err a = Left err | Right a

instance Monad (Either err) where
  return = Right
  Right va >>= k   = k va
  err      >>= _   = err
 -- Left verr >>=_ = Left verr
```

Monada **Either**(a excepțiilor)

```
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 (variantă simplificată)

```
newtype Writer log a = Writer {runWriter :: (a, log)}
-- a este parametru de tip
```

Monada Writer (variantă simplificată)

```haskell
newtype Writer log a = Writer {runWriter :: (a, log)}
-- a este parametru de tip

instance Monad (Writer String) where
  return va =  Writer (va, "")
  ma >>= k =   let (va, log1) = runWriter ma
                   (vb, log2) = runWriter (k va)
               in Writer (vb, log1 ++ log2)
```

Monada Writer (variantă simplificată)

```haskell
newtype Writer log a = Writer {runWriter :: (a, log)}
-- a este parametru de tip

instance Monad (Writer String) where
  return va =  Writer (va, "")
  ma >>= k =   let (va, log1) = runWriter ma
                   (vb, log2) = runWriter (k va)
               in Writer (vb, log1 ++ log2)


tell :: log -> Writer log ()
tell msg = Writer ((), msg)
```

Monada Writer - Exemplu logging

```haskell
newtype Writer log a = Writer {runWriter :: (a, log)}

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")
```

Monada Writer (varianta lungă)

Clasa de tipuri Semigroup

O mulțime, cu o operație <> care ar trebui să fie asociativă

```
class Semigroup a where
  (<>) :: a -> a -> a
```

Clasa de tipuri Monoid

Un semigrup cu unitatea mempty. mappend este alias pentru <>.

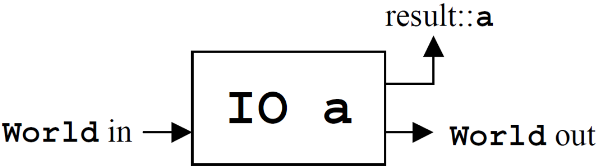
```
class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mappend = (<>)
```

Monada Writer

```haskell
newtype Writer log a = Writer {runWriter :: (a, log)}

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 **IO**

```
type IO a = RealWorld -> (a, RealWorld)
```



S. Peyton-Jones, Tackling the Awkward Squad: ...

Comenzi cu valori

- **IO** () corespunde comenzilor care nu produc rezultate

```
putChar  :: Char -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()
```

- În general, **IO** a corespunde comenzilor care produc rezultate de tip a.

```
getChar :: IO Char
getLine :: IO String
```

Compilarea programelor

Orice comandă **IO a** poate fi executată în interpretor, dar

Programele Haskell pot fi compilate

Fișierul scrie.hs:

```
main :: IO ()
main = putStrLn "?!"
```

```
08-io$ ghc scrie.hs
[1 of 1] Compiling Main  (scrie.hs, scrie.o)
Linking scrie.exe ...
08-io$ ./scrie
?!
```

Funcţia executată este **main**

Functor și Applicative definiți cu **return** și >>=

```haskell
instance Monad M where
  return a = ...
  ma >>= k = ...

instance Applicative M where
  pure = return
  mf <*> ma = do
    f <- mf
    a <- ma
    return (f a)
   -- mf >>= (\f -> ma >>= (\a -> return (f a)))

instance Functor F where         -- ma >>= \a -> return
    (f a)
  fmap f ma = pure f <*> ma      -- ma >>= (return . f)
```

Interpretoare monadice

Sintaxă abstractă

```
type Name = String

data Term = Var Name
          | Con Integer
          | Term :+: Term
          | Lam Name Term
          | App Term Term
```

Valori și medii de evaluare

```haskell
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

```haskell
test :: Term -> String
test t = showM $ interp t []
```

unde

```haskell
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
```

Interpretor 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 (runIndentity ma)
```

Interpretare în monada ‘Identity‘

Monada ‘Identity‘ este "efectul identitate".

```haskell
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
    return a           = Identity a
    ma >>= k = k (runIndentity ma)
```

Pentru a particulariza interpretorul definim

```haskell
type M a = Identity a

showM :: Show a => M a -> String
showM = show . runIdentity
```

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 (runIndentity ma)
```

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)

```haskell
data Maybe a = Nothing | Just a

instance Monad Maybe where
  return = Just
  Just a  >>= k    = k a
  Nothing >>= _    = Nothing
```

Putem renunța la valoarea ‘Wrong‘, folosind monada ‘Maybe‘

```haskell
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‘

```haskell
data Either a b = Left a | Right b

instance Monad (Either err) where
  return = Right
  Right a >>= k   = k a
  err     >>= _   = err
```

Putem nuanța erorile folosind monada ‘Either String‘

```haskell
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‘

```haskell
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.

```haskell
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)
```

Funcție ajutătoare

```haskell
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 șirul 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"
```

Monada listelor (a rezultatelor nedeterministe)

```
instance Monad [] where
  return va = [va]
  ma >>= k = [vb | va <- ma, vb <- k va]
```

Rezultatul nedeterminist e dat de lista tuturor valorilor posibile.

Monada listelor (a rezultatelor nedeterministe)

```haskell
instance Monad [] where
  return va = [va]
  ma >>= k = [vb | va <- ma, vb <- k va]
```

Rezultatul nedeterminist e dat de lista tuturor valorilor posibile.

```haskell
Prelude> [1,2,3] >>= \v -> [v+1]
[2,3,4]

Prelude> [1,2,3] >>= \v -> return (v+1)
[2,3,4]

Prelude> [4,9,25] >>= \x -> [(sqrt x), -(sqrt x)]
[2.0,-2.0,3.0,-3.0,5.0,-5.0]
```

Monada listelor (a rezultatelor nedeterministe)

```haskell
instance Monad [] where
  return va = [va]
  ma >>= k = [vb | va <- ma, vb <- k va]

radical :: Float -> [Float]
radical x | x >= 0 = [negate (sqrt x), sqrt x]
          | x < 0  = []

solEq2 :: Float -> Float -> Float -> [Float]
solEq2 0 0 c = []
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)
```

Interpretare în monada listelor

Adăugarea unei instrucțiuni nedeterministe

```
data Term = ... | Amb Term Term | Fail

type M a = [a]

showM :: Show a => M a -> String
showM = show

interp Fail _ = []
interp (Amb t1 t2) env = interp t1 env ++ interp t2
    env

pgmN = (App (Lam "x" (Var "x" :+: Var "x"))
       (Amb (Con 1) (Con2)))
> test pgmN
"[2,4]"
```

Pe săptămâna viitoare!

Curs 12

Introducere in 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 in λ-calcul.
- În 1935, independent de Church, Turing a dezvoltat mecanismul de calcul numit astăzi Mașina Turing. În 1936 și el a argumentat câ 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".

Referinţe

- 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: sintaxa

Lambda Calcul - sintaxă

$$
\begin{array}{rll}
t = & x & \text{(variabilă)} \\
 & \mid \lambda x.\, t & \text{(abstractizare)} \\
 & \mid t\ t & \text{(aplicare)}
\end{array}
$$

λ-calcul: sintaxa

Lambda Calcul - sintaxă

$$
\begin{array}{rll}
t = & x & \text{(variabilă)} \\
 & \mid \lambda x.\, t & \text{(abstractizare)} \\
 & \mid t\ t & \text{(aplicare)}
\end{array}
$$

λ-termeni

Fie $Var = \{x, y, z, \ldots\}$ o mulţime infinită de variabile.
Mulţimea 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 $x \in Var$ ş i $t \in \Lambda T$ atunci $(\lambda x.t) \in \Lambda T$

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), $(x(yx))$
- $(\lambda x.x)$, $(\lambda x.(xy))$, $(\lambda z.(xy))$, $(\lambda x.(\lambda z.(xy)))$
- $((\lambda x.x)y)$, $((\lambda x.(xz))y)$, $((\lambda x.x)(\lambda y.y))$

Convenţii:

- se elimină parantezele exterioare
- aplicarea este asociativă la stînga: $t_1 t_2 t_3$ 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 anonime

λ-termeni: exemple

- x, y, z
- (xy), (yx), $(x(yx))$
- $(\lambda x.x)$, $(\lambda x.(xy))$, $(\lambda z.(xy))$, $(\lambda x.(\lambda z.(xy)))$
- $((\lambda x.x)y)$, $((\lambda x.(xz))y)$, $((\lambda x.x)(\lambda y.y))$

Funcţii anonime în Haskell

În Haskell, `\` e folosit în locul simbolului λ și `->` în locul punctului.

$\lambda x.x * x$ este `\x -> x * x`

$\lambda x.x > 0$ este `\x -> x > 0`

Variabile libere şi legate

Apariţii libere şi legate

Pentru un termen $\lambda x.t$ spunem că:

- apariţiile variabilei x în t sunt legate (*bound*)
- λx este legătura (*binder*), iar t este domeniul (*scope*) legării
- o apariţie a unei variabile este liberă (*free*) dacă apare într-o poziţie în care nu e legată.

Un termen fără variable libere se numeşte 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 apariţie a lui x este liberă, a doua este legată.

Variabile libere

Mulţimea variabilelor libere $FV(t)$

Pentru un λ-termen t mulţimea 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:

$$
\begin{aligned}
FV(\lambda x.xy) = \quad & FV(xy) \setminus \{x\} \\
& (FV(x) \cup FV(y)) \setminus \{x\} \\
& (\{x\} \cup \{y\}) \setminus \{x\} \\
& \{y\}
\end{aligned}
$$

Variabile libere

Mulţimea variabilelor libere $FV(t)$

Pentru un λ-termen t mulţimea 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:

$$
\begin{aligned}
FV(\lambda x.xy) = \quad & FV(xy) \setminus \{x\} \\
& (FV(x) \cup FV(y)) \setminus \{x\} \\
& (\{x\} \cup \{y\}) \setminus \{x\} \\
& \{y\}
\end{aligned}
$$

$FV(x\lambda x.xy) =$

Variabile libere

Mulţimea variabilelor libere $FV(t)$

Pentru un λ-termen t mulţimea 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:

$$
\begin{aligned}
FV(\lambda x.xy) = \quad & FV(xy) \setminus \{x\} \\
& (FV(x) \cup FV(y)) \setminus \{x\} \\
& (\{x\} \cup \{y\}) \setminus \{x\} \\
& \{y\}
\end{aligned}
$$

$$FV(x\lambda x.xy) = \quad \{x, y\}$$

Substituţii

Fie t un λ-termen $x \in Var$.

Definiţie intuitivă

Pentru un λ-termen u vom nota prin $[u/x]t$ rezultatul înlocuirii tuturor apariţiilor libere ale lui x cu u în t.

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$
- $[(\lambda z.zw)/x](\lambda y.x) = \lambda y.\lambda z.zw$

Substituţii

Definirea substituţiei

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
$y \neq x$ ş i $y \notin FV(u)$

Substituţii

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$
- Cine este $[y/x]\lambda y.x$?

Substituţii

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 definiţia intuitivă obţinem
  $[y/x]\lambda y.x = \lambda y.y$ ceea ce este greşit!

Substituţii

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 definiţia intuitivă obţinem
  $[y/x]\lambda y.x = \lambda y.y$ ceea ce este greşit!

  Cum procedăm pentru a repara greşeala? Observăm că $\lambda y.x$ desemneaza o funcţie constantă, aceeaşi funcţie putând fi reprezentată prin $\lambda z.x$. Aplicarea corectă a substituţiei este:
  $[y/x]\lambda y.x = [y/x]\lambda z.x = \lambda z.y$

Substituţii

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 definiţia intuitivă obţinem
  $[y/x]\lambda y.x = \lambda y.y$ ceea ce este greşit!

  Cum procedăm pentru a repara greşeala? Observăm că $\lambda y.x$ desemneaza o funcţie constantă, aceeaşi funcţie putând fi reprezentată prin $\lambda z.x$. Aplicarea corectă a substituţiei este:
  $[y/x]\lambda y.x = [y/x]\lambda z.x = \lambda z.y$

  Avem libertatea de a redenumi variabilele legate!

α-conversie (α-echivalenţă)

α-conversia $=_\alpha$

[Reflexivitate] $t =_\alpha t$

[Simetrie] $t_1 =_\alpha t_2$ implică $t_2 =_\alpha t_1$

[Tranzitivitate] $t_1 =_\alpha t_2$ şi $t_2 =_\alpha t_3$ 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
$t\,t_1 =_\alpha t\,t_2$, $t_1 t =_\alpha t_2 t$ şi $\lambda x.t_1 =_\alpha \lambda x.t_2$

α-conversie (α-echivalenţă)

α-conversia $=_\alpha$

[Reflexivitate] $t =_\alpha t$

[Simetrie] $t_1 =_\alpha t_2$ implică $t_2 =_\alpha t_1$

[Tranzitivitate] $t_1 =_\alpha t_2$ şi $t_2 =_\alpha t_3$ 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

$tt_1 =_\alpha tt_2$, $t_1 t =_\alpha t_2 t$ şi $\lambda x.t_1 =_\alpha \lambda x.t_2$

$t_1 =_\alpha t_2$ şi $u_1 =_\alpha u_2$ implică $[u_1/x]t_1 =_\alpha [u_2/x]t_2$

Exemplu:
$[xy/x](\lambda y.yx) =_\alpha [xy/x](\lambda z.zx) =_\alpha \lambda z.z(xy)$

α-conversie (α-echivalenţă)

α-conversia $=_\alpha$

[Reflexivitate] $t =_\alpha t$

[Simetrie] $t_1 =_\alpha t_2$ implică $t_2 =_\alpha t_1$

[Tranzitivitate] $t_1 =_\alpha t_2$ şi $t_2 =_\alpha t_3$ 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

$tt_1 =_\alpha tt_2$, $t_1 t =_\alpha t_2 t$ şi $\lambda x.t_1 =_\alpha \lambda x.t_2$

$t_1 =_\alpha t_2$ şi $u_1 =_\alpha u_2$ implică $[u_1/x]t_1 =_\alpha [u_2/x]t_2$

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-echivalenţi vor fi consideraţi "egali".

α-conversie

Exemplu:

- $[xy/x]\lambda x.yx =_\alpha [xy/x]\lambda z.yz = \lambda z.[xy/x](yz) = \lambda z.yz$

  Observăm că $\lambda z.yz =_\alpha \lambda x.yx$

α-conversie

Exemplu:

- $[xy/x]\lambda x.yx =_\alpha [xy/x]\lambda z.yz = \lambda z.[xy/x](yz) = \lambda z.yz$

  Observăm că $\lambda z.yz =_\alpha \lambda x.yx$

α-conversie

Exemplu:

- $[xy/x]\lambda x.yx =_\alpha [xy/x]\lambda z.yz = \lambda z.[xy/x](yz) = \lambda z.yz$

  Observăm că $\lambda z.yz =_\alpha \lambda x.yx$

- $[y/z]\lambda xy.zzx = \lambda x.[y/z]\lambda y.zzx =_\alpha \lambda x.[y/z]\lambda v.zzx = \lambda x.\lambda v.[y/z](zzx) = \lambda xv.yyx$

β-reducţie

β-reducţia este o relaţie pe mulţimea α-termenilor.

β-reducţia $\to_\beta$, $\xrightarrow{*}_\beta$

- un singur pas $\to_\beta \subseteq \Lambda T \times \Lambda T$

  [Aplicarea] $(\lambda x.t)u \to_\beta [u/x]t$

  [Compatibilitatea] $t_1 \to_\beta t_2$ implică

  $t\, t_1 \to_\beta t\, t_2$, $t_1 t \to_\beta t_2 t$ şi $\lambda x.t_1 \to_\beta \lambda x.t_2$

- zero sau mai mulţi paşi $\xrightarrow{*}_\beta \subseteq \Lambda T \times \Lambda T$

  $t_1 \xrightarrow{*}_\beta t_2$ dacă există $n \geq 0$ şi $u_0, \ldots, u_n$ astfel încât

  $t_1 =_\alpha u_0 \to_\beta u_1 \to_\beta \cdots \to_\beta u_n =_\alpha t_2$

β-reducţie

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-reducţie

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

- $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda y.yv)z \to_\beta zv$
- $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda x.zx)v \to_\beta zv$

β-reducţie

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

- $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda y.yv)z \to_\beta zv$
- $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda x.zx)v \to_\beta zv$

Observăm că un termen poate fi β-redus în mai multe moduri.

Proprietatea de confluenţă ne asigură că vom ajunge întotdeauna la acelaşi rezultat.

Confluenţa β-reducţiei

Teorema Church-Rosser

Dacă $t \xrightarrow{*}_{\beta} t_1$ şi $t \xrightarrow{*}_{\beta} t_2$

$$
\begin{tikzcd}
 & t \arrow[dl, "*"'] \arrow[dr, "*"] & \\
t_1 & & t_2
\end{tikzcd}
$$

Confluenţa β-reducţiei

Teorema Church-Rosser

Dacă $t \xrightarrow{*}_\beta t_1$ şi $t \xrightarrow{*}_\beta t_2$

$$\begin{tikzcd}
 & t \arrow[dl, "*"'] \arrow[dr, "*"] & \\
t_1 & & t_2
\end{tikzcd}$$

atunci există u astfel încât $t_1 \xrightarrow{*}_\beta u$ şi $t_2 \xrightarrow{*}_\beta u$.

$$\begin{tikzcd}
 & t \arrow[dl, "*"'] \arrow[dr, "*"] & \\
t_1 \arrow[dr, "*"'] & & t_2 \arrow[dl, "*"] \\
 & u &
\end{tikzcd}$$

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 $\to_\beta$ se numeş te *β-formă normală*
- dacă $t \stackrel{*}{\to}_\beta u_1$, $t \stackrel{*}{\to}_\beta u_2$ ş i u_1, u_2 sunt β-forme normale atunci, datorită confluenţei, $u_1 =_\alpha u_2$
- un λ-termen poate avea cel mult o β-formă normală (modulo α-echivalenţă)

β-forma normală

Formă normal ă

- un λ-termen căruia nu i se mai poate aplica reducerea într-un pas $\to_\beta$ se numeş te *β-formă normală*
- dacă $t \xrightarrow{*}_\beta u_1$, $t \xrightarrow{*}_\beta u_2$ ş i u_1, u_2 sunt η-forme normale atunci, datorită confluenţei, $u_1 =_\alpha u_2$
- un λ-termen poate avea cel mult o β-formă normală (modulo α-echivalenţă)

Exemplu:

- zv este β-formă normală pentru $(\lambda x.(\lambda y.yx)z)v$
  $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda y.yv)z \to_\beta zv$
- există termeni care **nu** pot fi reduş i la o β-formă normală, de exemplu $(\lambda x.xx)(\lambda x.xx)$

β-conversia

Intuitiv, β-conversia extinde β-reducţia în ambele direcţii.

- $(\lambda y.yv)z \rightarrow_\beta zv \leftarrow_\beta (\lambda x.zx)v$

β-conversia

Intuitiv, β-conversia extinde β-reducţia în ambele direcţii.

- $(\lambda y.yv)z \to_\beta zv \leftarrow_\beta (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_\beta (\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda x.zx)v$

β-conversia

Intuitiv, β-conversia extinde β-reducţia în ambele direcţii.

- $(\lambda y.yv)z \to_\beta zv \leftarrow_\beta (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_\beta (\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda x.zx)v$

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ ş i $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ ş i, pentru orice i, $u_i \to_\beta u_{i+1}$ sau $u_{i+1} \to_\beta u_i$

β-conversia

Intuitiv, β-conversia extinde β-reducţia în ambele direcţii.

- $(\lambda y.yv)z \to_\beta zv \leftarrow_\beta (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_\beta (\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda x.zx)v$

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ ş i $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ ş i, pentru orice i, $u_i \to_\beta u_{i+1}$ sau $u_{i+1} \to_\beta u_i$

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 $n \geq 0$ ş i $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ ş i, pentru orice i, $u_i \to_\beta u_{i+1}$ sau $u_{i+1} \to_\beta u_i$

β-conversia

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ ş i $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ ş i, pentru orice i, $u_i \to_\beta u_{i+1}$ sau $u_{i+1} \to_\beta u_i$

Observaţii

- $=_\beta$ este o relaţie de echivalenţă

β-conversia

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ şi $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ şi, pentru orice i, $u_i \to_\beta u_{i+1}$ sau $u_{i+1} \to_\beta u_i$

Observaţii

- $=_\beta$ este o relaţie de echivalenţă
- pentru t_1, t_2 λ-termeni şi u_1, u_2 β-forme normale
  dacă $t_1 \stackrel{*}{\to}_\beta u_1$, $t_2 \stackrel{*}{\to}_\beta u_2$ şi $u_1 =_\alpha u_2$ 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 $n \geq 0$ ş i $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ ş i, pentru orice i, $u_i \to_\beta u_{i+1}$ sau $u_{i+1} \to_\beta u_i$
- $=_\beta$ este o relaţie de echivalenţă

β-conversia

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ ş i $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ ş i, pentru orice i, $u_i \rightarrow_\beta u_{i+1}$ sau $u_{i+1} \rightarrow_\beta u_i$
- $=_\beta$ este o relaţie de echivalenţă
- pentru t_1, t_2 λ-termeni ş i u_1, u_2 β-forme normale
  dacă $t_1 \overset{*}{\rightarrow}_\beta u_1$, $t_2 \overset{*}{\rightarrow}_\beta u_2$ şi $u_1 =_\alpha u_2$ atunci $t_1 =_\beta t_2$

β-conversia reprezintă "egalitatea prin calcul", iar β-reducţia (modulo α-conversie) oferă o procedură de decizie pentru aceasta.

Codificări în λ-calcul

Ideea generală

Intuiție

Tipurile de date sunt codificate de capabilități

Boole	capabilitatea de a alege între două alternative
Perechi	capabilitatea de a calcula ceva bazat pe două valori
Numere naturale	capabilitatea de a itera de un număr dat de ori

Valori de adevăr

Intuiție: Capabilitatea de a alege între două alternative.

Codificare: Un Boolean este o funcție cu 2 argumente reprezentând ramurile unei alegeri.

true ::= $\lambda t\ f.t$ — din cele două alternative o alege pe prima

false ::= $\lambda t\ f.f$ — din cele două alternative o alege pe a doua

Operații Booleene

true ::= $\lambda t\ f.t$ — din cele două alternative o alege pe prima

false ::= $\lambda t\ f.f$ — din cele două alternative o alege pe a doua

`if` ::= $\lambda c\ t\ e.c\ t\ e$ — pur și simplu folosim valoarea de adevăr pentru a alege între alternative

Operații Booleene

$$\textit{true} ::= \lambda t\ f.t$$ — din cele două alternative o alege pe prima

$$\textit{false} ::= \lambda t\ f.f$$ — din cele două alternative o alege pe a doua

$$\texttt{if} ::= \lambda c\ t\ e.c\ t\ e$$ — pur și simplu folosim valoarea de adevăr pentru a alege între alternative

$$\texttt{if}\ \textit{false}\ (\lambda x.x\ x)\ (\lambda x.x) \rightarrow_\beta^3$$
$$\textit{false}\ (\lambda x.x\ x)\ (\lambda x.x) \rightarrow_\beta^2 \lambda x.x$$

Operații Booleene

true ::= $\lambda t\ f.t$ — din cele două alternative o alege pe prima

false ::= $\lambda t\ f.f$ — din cele două alternative o alege pe a doua

`if` ::= $\lambda c\ t\ e.c\ t\ e$

`and` ::= $\lambda b1\ b2.$ `if` $b1\ b2$ *false* sau $\lambda b1\ b2.b1\ b2\ b1$

`and` *true false*

Operații Booleene

$true ::= \lambda t\ f.t$ — din cele două alternative o alege pe prima

$false ::= \lambda t\ f.f$ — din cele două alternative o alege pe a doua

$\texttt{if} ::= \lambda c\ t\ e.c\ t\ e$

$\texttt{and} ::= \lambda b1\ b2.\ \texttt{if}\ b1\ b2\ false$ sau $\lambda b1\ b2.b1\ b2\ b1$

$\texttt{and}\ true\ false \rightarrow_\beta^2 true\ false\ true \rightarrow_\beta^2 false$

$\texttt{or} ::= \lambda b1\ b2.\ \texttt{if}\ b1\ true\ b2$ sau $\lambda b1\ b2.b1\ b1\ b2$

$\texttt{or}\ true\ false$

Operații Booleene

$true ::= \lambda t\ f.t$ — din cele două alternative o alege pe prima

$false ::= \lambda t\ f.f$ — din cele două alternative o alege pe a doua

$\texttt{if} ::= \lambda c\ t\ e.c\ t\ e$

$\texttt{and} ::= \lambda b1\ b2.\ \texttt{if}\ b1\ b2\ false$ sau $\lambda b1\ b2.b1\ b2\ b1$

$\texttt{and}\ true\ false \rightarrow_{\beta}^{2} true\ false\ true \rightarrow_{\beta}^{2} false$

$\texttt{or} ::= \lambda b1\ b2.\ \texttt{if}\ b1\ true\ b2$ sau $\lambda b1\ b2.b1\ b1\ b2$

$\texttt{or}\ true\ false \rightarrow_{\beta}^{2} true\ true\ false \rightarrow_{\beta}^{2} true$

$\texttt{not} ::= \lambda b.\ \texttt{if}\ b\ false\ true$ sau $\lambda b\ t\ f.b\ f\ t$

$\texttt{not}\ true \rightarrow_{\beta} \lambda t\ f.true\ f\ t \rightarrow_{\beta} \lambda t\ f.f$

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 si apoi lui y.

Operații pe perechi

$$\texttt{pair} ::= \lambda x\, y.\lambda f.f\ x\ y$$

$$\texttt{pair}\ x\ y \equiv_\beta f\ x\ y$$

$$\texttt{fst} ::= \lambda p.p\ \mathit{true} \text{ — } \mathit{true} \text{ alege prima componentă}$$

$$\texttt{fst}\ (\texttt{pair}\ x\ y) \rightarrow_\beta \texttt{pair}\ x\ y\ \mathit{true} \rightarrow_\beta^3 \mathit{true}\ x\ y \rightarrow_\beta^2 x$$

$$\texttt{snd} ::= \lambda p.p\ \mathit{false} \text{ — } \mathit{false} \text{ alege a doua componentă}$$

$$\texttt{snd}\ (\texttt{pair}\ x\ y) \rightarrow_\beta \texttt{pair}\ x\ y\ \mathit{false} \rightarrow_\beta^3 \mathit{false}\ x\ y \rightarrow_\beta^2 y$$

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 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

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 ::= $\lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

...

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 ::= $\lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

...

Observație: $0 = \mathit{false}$

Operații aritmetice de bază

$0 ::= \lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

$8 ::= \lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

$S ::= \lambda n\ s\ z.s\ (n\ s\ z)$ sau $\lambda n\ s\ z.n\ s\ (sz)$

$S\ 0$

Operații aritmetice de bază

$0 ::= \lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

$8 ::= \lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

$S ::= \lambda n\ s\ z.s\ (n\ s\ z)$ sau $\lambda n\ s\ z.n\ s\ (sz)$

$$S\ 0 \rightarrow_{\beta} \lambda s\ z.0s(sz) \rightarrow_{\beta}^{2} \lambda s\ z.sz = 1$$

Operații aritmetice de bază

$0 ::= \lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

$8 ::= \lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

$S ::= \lambda n\ s\ z.s\ (n\ s\ z)$ sau $\lambda n\ s\ z.n\ s\ (sz)$

$S\ 0 \rightarrow_\beta \lambda s\ z.0s(sz) \rightarrow_\beta^2 \lambda s\ z.sz = 1$

Observăm că $m\ s$ aplică funcţia s de m ori.

Operații aritmetice de bază

$0 ::= \lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

$8 ::= \lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

$S ::= \lambda n\ s\ z.s\ (n\ s\ z)$ sau $\lambda n\ s\ z.n\ s\ (sz)$

$S\ 0 \to_\beta \lambda s\ z.0s(sz) \to_\beta^2 \lambda s\ z.sz = 1$

Observăm că $m\ s$ aplică funcţia s de m ori.

$+ ::= \lambda m\ n.(m\ \ S)\ n$ sau $\lambda m\ n.\lambda s\ z.m\ s\ (n\ s\ z)$

$+\ 3\ 2$

Operații aritmetice de bază

$0 ::= \lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

$8 ::= \lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

$\mathsf{S} ::= \lambda n\ s\ z.s\ (n\ s\ z)$ sau $\lambda n\ s\ z.n\ s\ (sz)$

$\mathsf{S}\ 0 \rightarrow_\beta \lambda s\ z.0s(sz) \rightarrow_\beta^2 \lambda s\ z.sz = 1$

Observăm că $m\ s$ aplică funcţia s de m ori.

$+ ::= \lambda m\ n.(m\ \ \mathsf{S})\ n$ sau $\lambda m\ n.\lambda s\ z.m\ s\ (n\ s\ z)$

$+\ 3\ 2 \rightarrow_\beta^2 \lambda s\ z.3\ s\ (2\ s\ z) \rightarrow_\beta^2$
$\lambda s\ z.s(s(s(2\ s\ z))) \rightarrow_\beta^2 \lambda s\ z.s(s(s(s(s\ z)))) = 5$

Operații aritmetice de bază

0 ::= $\lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

8 ::= $\lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

S ::= $\lambda n\ s\ z.s\ (n\ s\ z)$ sau $\lambda n\ s\ z.n\ s\ (sz)$

$\mathsf{S}\ 0 \rightarrow_\beta \lambda s\ z.0s(sz) \rightarrow^2_\beta \lambda s\ z.sz = 1$

Observăm că $m\ s$ aplică funcţia s de m ori.

$+$ ::= $\lambda m\ n.(m\ \ \mathsf{S})\ n$ sau $\lambda m\ n.\lambda s\ z.m\ s\ (n\ s\ z)$

$+\ 3\ 2 \rightarrow^2_\beta \lambda s\ z.3\ s\ (2\ s\ z) \rightarrow^2_\beta$
$\lambda s\ z.s(s(s(2\ s\ z))) \rightarrow^2_\beta \lambda s\ z.s(s(s(s(s\ z))) = 5$

$*$ ::= $\lambda m\ n.m\ (+\ n)\ 0$ sau $\lambda m\ n.\lambda s.m\ (n\ s)$

$*\ 3\ 2$

Operații aritmetice de bază

$0 ::= \lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

$8 ::= \lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

$S ::= \lambda n\ s\ z.s\ (n\ s\ z)$ sau $\lambda n\ s\ z.n\ s\ (sz)$

$S\ 0 \rightarrow_\beta \lambda s\ z.0s(sz) \rightarrow^2_\beta \lambda s\ z.sz = 1$

Observăm că $m\ s$ aplică funcţia s de m ori.

$+ ::= \lambda m\ n.(m\ \ S)\ n$ sau $\lambda m\ n.\lambda s\ z.m\ s\ (n\ s\ z)$

$+\ 3\ 2 \rightarrow^2_\beta \lambda s\ z.3\ s\ (2\ s\ z) \rightarrow^2_\beta$
$\lambda s\ z.s(s(s(2\ s\ z))) \rightarrow^2_\beta \lambda s\ z.s(s(s(s(s\ z)))) = 5$

$* ::= \lambda m\ n.m\ (+\ n)\ 0$ sau $\lambda m\ n.\lambda s.m\ (n\ s)$

$*\ 3\ 2 \rightarrow^2_\beta 3\ (+\ 2)0 \rightarrow^2_\beta +\ 2(+\ 2(+\ 2\ 0)) \rightarrow^4_\beta$
$+\ 2(+\ 2\ 2) \rightarrow^4_\beta +\ 2\ 4 \rightarrow^4_\beta 6$

Liste

Intuiție: Capabilitatea de a agrega o listă

Codificare: O funcție cu 2 argumente:
funcția de agregare și *valoarea inițială*
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 inițială*
Lista $[3, 5]$ este reprezentată prin $a\ 3\ (a\ 5\ i)$

$\texttt{null}$ ::= $\lambda a\ i.i$ — lista vidă

$\texttt{cons}$ ::= $\lambda x\ l.\lambda a\ i.a\ x\ (l\ a\ i)$
Constructorul de liste

Exemplu: $\texttt{cons}\ 3\ (\texttt{cons}\ 5\ \texttt{null}) \rightarrow_{\beta}^{2} \lambda a\ i.a\ 3\ (\texttt{cons}\ 5\ \texttt{null}\ a\ i) \rightarrow_{\beta}^{4} \lambda a\ i.a\ 3\ (a\ 5\ (\texttt{null}\ a\ i)) \rightarrow_{\beta}^{2} \lambda a\ i.a\ 3\ (a\ 5\ i)$

Lista $[3, 5]$ reprezintă capabilitatea de a agrega elementele 3 si apoi 5 dată fiind o funcție de agregare a și o valoare implicită i.

Operații pe liste

`null` ::= $\lambda a\ i.i$ — lista vidă

`cons` ::= $\lambda x\ l.\lambda a\ i.a\ x\ (l\ a\ i)$

Operații pe liste

`null` ::= $\lambda a\ i.i$ — lista vidă

`cons` ::= $\lambda x\ l.\lambda a\ i.a\ x\ (l\ a\ i)$

`?null` ::= $\lambda l.l\ (\lambda x\ v.false)\ true$

Operații pe liste

$\texttt{null} ::= \lambda a\ i.i$ — lista vidă

$\texttt{cons} ::= \lambda x\ l.\lambda a\ i.a\ x\ (l\ a\ i)$

$\texttt{?null} ::= \lambda l.l\ (\lambda x\ v.\mathit{false})\ \mathit{true}$

$\texttt{head} ::= \lambda d\ l.l\ (\lambda x\ v.x)\ d$

primul element al listei, sau *d* dacă lista e vidă

Operații pe liste

$\texttt{null} ::= \lambda a\ i.i$ — lista vidă

$\texttt{cons} ::= \lambda x\ l.\lambda a\ i.a\ x\ (l\ a\ i)$

$\texttt{?null} ::= \lambda l.l\ (\lambda x\ v.false)\ true$

$\texttt{head} ::= \lambda d\ l.l\ (\lambda x\ v.x)\ d$
primul element al listei, sau d dacă lista e vidă

$\texttt{tail} ::= \lambda l.\ \texttt{fst}\ (l\ (\lambda x\ p.\ \texttt{pair}\ (\texttt{snd}\ p)\ (\texttt{cons}\ x\ (\texttt{snd}\ p)))\ (\texttt{pair}\ \texttt{null}\ \texttt{null}))$
coada listei, sau lista vidă dacă lista e vidă

Liste ca perechi

Intuiție: putem reprezenta o lista ca o pereche formată din primul element si restul listei

Lista vidă: folosim şi o valoare booleana care indică dacă lista este vidă sau nevidă

Lista $[3, 5]$ este reprezentată prin

$\mathit{pair}\ \mathit{false}\ (\mathit{pair}\ 3\ (\mathit{pair}\ \mathit{false}\ (\mathit{pair}\ 5\ (\mathit{pair}\ \mathit{true}\ \mathit{true}))))$

Liste ca perechi

Intuiție: putem reprezenta o lista ca o pereche formată din primul element si restul listei

Lista vidă: folosim şi o valoare booleana care indică dacă lista este vidă sau nevidă
Lista $[3, 5]$ este reprezentată prin
$\mathit{pair}\ \mathit{false}\ (\mathit{pair}\ 3\ (\mathit{pair}\ \mathit{false}\ (\mathit{pair}\ 5\ (\mathit{pair}\ \mathit{true}\ \mathit{true}))))$

Liste ca perechi

Intuiție: putem reprezenta o lista ca o pereche formată din primul element si restul listei

Lista vidă: folosim şi o valoare booleana care indică dacă lista este vidă sau nevidă
Lista $[3, 5]$ este reprezentată prin
pair false (*pair* 3 (*pair false* (*pair* 5 (*pair true true*))))

`null` ::= `pair` *true true* — lista vidă

`?null` ::= `fst`

Liste ca perechi

Intuiție: putem reprezenta o lista ca o pereche formată din primul element si restul listei

Lista vidă: folosim şi o valoare booleana care indică dacă lista este vidă sau nevidă
Lista $[3, 5]$ este reprezentată prin
$\mathit{pair}\ \mathit{false}\ (\mathit{pair}\ 3\ (\mathit{pair}\ \mathit{false}\ (\mathit{pair}\ 5\ (\mathit{pair}\ \mathit{true}\ \mathit{true}))))$

`null` ::= `pair` *true true* — lista vidă

`?null` ::= `fst`

`cons` ::= $\lambda x\, l.$ `pair` *false* (`pair` $x\ l$)

Liste ca perechi

Intuiție: putem reprezenta o lista ca o pereche formată din primul element si restul listei

Lista vidă: folosim şi o valoare booleana care indică dacă lista este vidă sau nevidă
Lista $[3, 5]$ este reprezentată prin
pair false (*pair* 3 (*pair false* (*pair* 5 (*pair true true*))))

$\texttt{null} ::= \texttt{pair}\ \ \textit{true true}$ — lista vidă

$\texttt{?null} ::= \texttt{fst}$

$\texttt{cons} ::= \lambda x\ l.\ \texttt{pair}\ \ \textit{false}\ (\texttt{pair}\ \ x\ l)$

$\texttt{head} ::= \lambda l.\ \texttt{fst}\ \ (\texttt{snd}\ \ l)$

$\texttt{tail} ::= \lambda l.\ \texttt{snd}\ \ (\texttt{snd}\ \ l)$

Pe săptămâna viitoare!

Curs 13

Semantica small-step pentru λ-calcul

Sintaxa limbajului LAMBDA

BNF

$$
\begin{array}{rcl}
e & ::= & x \mid n \mid true \mid false \\
 & \mid & e + e \mid e < e \mid \texttt{not}\ (e) \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & \mid & \lambda x.e \mid e\ e \\
 & \mid & \texttt{let}\ x = e\ \texttt{in}\ e
\end{array}
$$

Sintaxa limbajului LAMBDA

BNF

$$
\begin{array}{lcl}
e & ::= & x \mid n \mid \mathit{true} \mid \mathit{false} \\
 & \mid & e + e \mid e < e \mid \texttt{not}\ (e) \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & \mid & \lambda x.e \mid e\ e \\
 & \mid & \texttt{let}\ x = e\ \texttt{in}\ e
\end{array}
$$

Verificarea sintaxei în Prolog

```
exp(Id) :- atom(Id).          %identificator
exp(Lit) :- Lit = true ; Lit = false ; integer(Lit).
exp(E1 + E2) :- exp(E1), exp(E2).
exp(if(E1, E2, E3)) :- exp(E1), exp(E2), exp(E3).
exp(Id -> Exp) :- atom(Id), exp(Exp).     % lambda
exp(Exp1 $ Exp2) :- exp(Exp1), exp(Exp2). % aplicare
exp(let(Id, Exp1, Exp2)) :- atom(Id), exp(Exp1), exp(Exp2).
```

Semantica small-step pentru Lambda

- Definește cel mai mic pas de execuție ca o relație de tranziție între expresii dată fiind o stare cu valori pentru variabilele libere

  $\rho \vdash cod \to cod'$ `step(Env, Cod1,Cod2)`

- Mediul de evaluare ρ este format din perechi variabilă-valoare (x, v) unde variabilele sunt reprezentate prin identificatori, iar valorile sunt *întregi*, *booleene*, sau *valori funcție*.

- Execuția se obține ca o succesiune de astfel de tranziții.

Semantica variabilelor

$$\rho \vdash x \to v \quad \textit{dacă}\ \rho(x) = v$$

Prolog

```
step(Env, X, V) :- atom(X), get(Env, X, V).
```

Semantica expresiilor aritmetice

- Semantica adunării a două expresii aritmetice

  $\langle i_1 + i_2 , \sigma\rangle \to \langle i , \sigma\rangle \quad \textit{dacă } i = i_1 + i_2$

  $$\frac{\langle a_1 , \sigma\rangle \to \langle a_1' , \sigma'\rangle}{\langle a_1 + a_2 , \sigma\rangle \to \langle a_1' + a_2 , \sigma'\rangle} \qquad \frac{\langle a_2 , \sigma\rangle \to \langle a_2' , \sigma'\rangle}{\langle a_1 + a_2 , \sigma\rangle \to \langle a_1 + a_2' , \sigma'\rangle}$$

- Pentru alți operatori (artimetici, de comparație, booleeni, condițional)
  - Similar cu regulile din IMP

Semantica expresiilor aritmetice

- Semantica adunării a două expresii aritmetice

$\langle i_1 + i_2 \, , \, \sigma \rangle \to \langle i \, , \, \sigma \rangle \quad$ *dacă* $i = i_1 + i_2$

$$\frac{\langle a_1 \, , \, \sigma \rangle \to \langle a_1' \, , \, \sigma' \rangle}{\langle a_1 + a_2 \, , \, \sigma \rangle \to \langle a_1' + a_2 \, , \, \sigma' \rangle} \qquad \frac{\langle a_2 \, , \, \sigma \rangle \to \langle a_2' \, , \, \sigma' \rangle}{\langle a_1 + a_2 \, , \, \sigma \rangle \to \langle a_1 + a_2' \, , \, \sigma' \rangle}$$

Prolog

```
step(_, I1 + I2,I):- integer(I1),integer(I2),
                     I is I1 + I2.

step(Env, AE + AE1,AE + AE2):- step(Env, AE1,AE2).

step(Env, AE1 + AE,AE2 + AE):- step(Env, AE1,AE2).
```

Semantica λ-abstracției

$$\rho \vdash \lambda x.e \rightarrow closure(x, e, \rho)$$

λ-abstracția se evaluează la o valoare specială numită closure care capturează valorile curente ale variabilelor pentru a se putea executa în acest mediu atunci când va fi aplicată.

Prolog

```
step(Env, X -> E, closure(X, E, Env)).
```

Semantica construcției `let`

$$\rho \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightarrow (\lambda x.e2)\ e_1$$

A îi da lui x valoarea lui e_1 în e_2 este același lucru cu a aplica funcția de x cu corpul e_2 expresiei e_1.

Prolog

```
step(_, let(X, E1, E2), (X -> E2) $ E1).
```

Semantica operatorului de aplicare

$$\frac{\rho_{e_{x \leftarrow v}} \vdash e \rightarrow e'}{\rho \vdash closure(x, e, \rho_e)\ v \rightarrow closure(x, e', \rho_e)\ v} \quad \textit{dacă } v \text{ valoare}$$

$$\rho \vdash closure(x, e, \rho_e)\ v \rightarrow e \quad \textit{dacă } e \text{ valoare}$$

$$\frac{\rho \vdash e_1 \rightarrow e_1'}{\rho \vdash e_1\ e_2 \rightarrow e_1'\ e_2}$$

$$\frac{\rho \vdash e_2 \rightarrow e_2'}{\rho \vdash e_1\ e_2 \rightarrow e_1\ e_2'}$$

Semantica operatorului de aplicare

$$\frac{\rho_{e_{x \leftarrow v}} \vdash e \to e'}{\rho \vdash \mathit{closure}(x, e, \rho_e)\ v \to \mathit{closure}(x, e', \rho_e)\ v} \quad \textit{dacă } v \text{ valoare}$$

$$\rho \vdash \mathit{closure}(x, e, \rho_e)\ v \to e \quad \textit{dacă } e \text{ valoare}$$

$$\frac{\rho \vdash e_1 \to e_1'}{\rho \vdash e_1\ e_2 \to e_1'\ e_2} \qquad \frac{\rho \vdash e_2 \to e_2'}{\rho \vdash e_1\ e_2 \to e_1\ e_2'}$$

Prolog

```
step(Env, E $ E1, E $ E2) :- step(Env, E1, E2).
step(Env, E1 $ E, E2 $ E) :- step(Env, E1, E2).
step(Env, closure(X, E, EnvE) $ V, Result) :-
      set(EnvE, X, V, EnvEX),
      step(EnvEX, E, E1)  ->
              Result = closure(X, E1, EnvE) $ V
              ;  ( Result = E).
```

Semantica small-step: toți pașii

Prolog

```
all_steps(Env, E1, Trace, E) :-
    step(Env, E1, E2)
    ->  all_steps(Env, E2, Trace1, E), Trace = [E2|Trace1]
    ;   E = E1, Trace=[].

run(E, V) :- all_steps([], E, _, V).
```

Exemplu

```
lam3((x -> ((y -> (x + y)) $ 7)) $ 3).

?- lam3(X), run(X,V).
X =  (x->(y->x+y)$7)$3,
V = 10
```

Semantica small-step: toți pașii

Prolog

```
all_steps(Env, E1, Trace, E) :-
    step(Env, E1, E2)
    ->  all_steps(Env, E2, Trace1, E), Trace = [E2|Trace1]
    ;   E = E1, Trace=[].

print_list([]).
print_list([H|T]) :- print(H), nl, print_list(T).

trace(E1) :- all_steps([], E1, Trace, _), print_list(Trace).
```

Semantica small-step: toți pașii

Exemplu

```
lam3((x -> ((y -> (x + y)) $ 7)) $ 3).
```

```
?- lam3(X), trace(X).
closure(x,(y->x+y)$7,[])$3
closure(x,closure(y,x+y,[(x,3)])$7,[])$3
closure(x,closure(y,3+y,[(x,3)])$7,[])$3
closure(x,closure(y,3+7,[(x,3)])$7,[])$3
closure(x,closure(y,10,[(x,3)])$7,[])$3
closure(x,10,[])$3
10
X =  (x->(y->x+y)$7)$3.
```

Pe săptămâna viitoare!

Curs 14

Cuprins

1. Determinarea tipurilor
   - Asociere de tipuri

Sintaxa limbajului LAMBDA

BNF

$$
\begin{array}{rcl}
e & ::= & x \mid n \mid \mathit{true} \mid \mathit{false} \\
 & \mid & e + e \mid e < e \mid \mathtt{not}\,(e) \\
 & \mid & \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \\
 & \mid & \lambda x.e \mid e\ e \\
 & \mid & \mathtt{let}\ x = e\ \mathtt{in}\ e
\end{array}
$$

Verificarea sintaxei în Prolog

```
exp(Id) :- atom(Id).                              % identificator
exp(Lit) :- Lit = true ; Lit = false ; integer(Lit).
exp(E1 + E2) :- exp(E1), exp(E2).
exp(if(E1, E2, E3)) :- exp(E1), exp(E2), exp(E3).
exp(Id -> Exp) :- atom(Id), exp(Exp).             % lambda
exp(Exp1 $ Exp2) :- exp(Exp1), exp(Exp2).         % aplicare
exp(let(Id, Exp1, Exp2)) :- atom(Id), exp(Exp1), exp(Exp2).
```

Semantica small-step pentru Lambda

Prolog

```
step(Env, X, V) :- atom(X), get(Env, X, V).
step(_, I1 + I2,I):- integer(I1),integer(I2),
                     I is I1 + I2.
step(Env, AE + AE1,AE + AE2):- step(Env, AE1,AE2).
step(Env, AE1 + AE,AE2 + AE):- step(Env, AE1,AE2).
step(Env, X -> E, closure(X, E, Env)).
step(_, let(X, E1, E2), (X -> E2) $ E1).
step(Env, E $ E1, E $ E2) :- step(Env, E1, E2).
step(Env, E1 $ E, E2 $ E) :- step(Env, E1, E2).
step(Env, closure(X, E, EnvE) $ V, Result) :-
        set(EnvE, X, V, EnvEX),
        step(EnvEX, E, E1)
        ->  Result = closure(X, E1, EnvE) $ V
        ;   Result = E.
```

Problemă: Sintaxa este prea permisivă

Problemă: Mulți termeni acceptați de sintaxă nu pot fi evaluați

- $2\ (\lambda x.x)$— expresia din stânga aplicației trebuie să reprezinte o funcție
- $(\lambda x.x) + 1$ — adunăm funcții cu numere
- $(\lambda x.x + 1)\ (\lambda x.x)$ — pot face o reducție, dar tot nu pot evalua

```
?- run((x -> x) +1, V).
V = closure(x, x, [])+1.

?- run(2 $ (x -> x), V).
V = 2$closure(x, x, []).
```

Problemă: Sintaxa este prea permisivă

Problemă: Mulți termeni acceptați de sintaxă nu pot fi evaluați

- 2 $(\lambda x.x)$— expresia din stânga aplicației trebuie să reprezinte o funcție
- $(\lambda x.x) + 1$ — adunăm funcții cu numere
- $(\lambda x.x + 1)\ (\lambda x.x)$ — pot face o reducție, dar tot nu pot evalua

Soluție: Identificarea (precisă) a programelor corecte

- Definim tipuri pentru fragmente de program corecte (e.g., int, bool)
- Definim (recursiv) o relație care să lege fragmente de program de tipurile asociate

$$((\lambda x.x + 1)\ ((\lambda x.x)\ 3)) : \text{int}$$

Relația de asociere de tipuri

Definim (recursiv) o relație de forma $\Gamma \vdash e : \tau$, unde

- τ este un tip

  $\tau ::= int$ [întregi]
  $\quad\mid bool$ [valori de adevăr]
  $\quad\mid \tau \to \tau$ [funcții]
  $\quad\mid a$ [variabile de tip]

- e este un termen (potențial cu variabile libere)
- Γ este mediul de tipuri, o funcție parțială finită care asociază tipuri variabilelor (libere ale lui e)
- Variabilele de tip sunt folosite pentru a indica polimorfismul

Cum citim $\Gamma \vdash e : \tau$?

Dacă variabila x are tipul $\Gamma(x)$ pentru orice $x \in dom(\Gamma)$,
atunci termenul e are tipul τ.

Axiome

(:VAR) $\Gamma \vdash x : \tau$ *dacă* $\Gamma(x) = \tau$

(:INT) $\Gamma \vdash n : int$ *dacă* n întreg

(:BOOL) $\Gamma \vdash b : bool$ *dacă* $b = true$ *or* $b = false$

Expresii

$$(\text{:IOP})\quad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ o\ e_2 : int} \quad \textit{dacă } o \in \{+, -, *, /\}$$

$$(\text{:COP})\quad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ o\ e_2 : bool} \quad \textit{dacă } o \in \{\leq, \geq, <, >, =\}$$

$$(\text{:BOP})\quad \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1\ o\ e_2 : bool} \quad \textit{dacă } o \in \{,\}$$

$$(\text{:IF})\quad \frac{\Gamma \vdash e_b : bool \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e_b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

Fragmentul funcțional

$$(:\text{FN})\quad \frac{\Gamma' \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \quad \textit{dacă } \Gamma' = \Gamma[x \mapsto \tau]$$

$$(:\text{APP})\quad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

Probleme computaționale

Verificarea tipului

Date fiind Γ, e și τ, verificați dacă $\Gamma \vdash e : \tau$.

Determinarea (inferarea) tipului

Date fiind Γ și e, găsiți (sau arătați ce nu există) un τ astfel încât $\Gamma \vdash e : \tau$.

- A doua problemă e mai grea în general decât prima
- Algoritmi de inferare a tipurilor
  - Colectează constrângeri asupra tipului
  - Folosesc metode de rezolvare a constrângerilor (programare logică)

Exemplu

Care este tipul expresiei următoare (dacă are)

$$\lambda x.\lambda y.\lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y$$

Aplicăm regula

$$(\text{:FN})\quad \frac{\Gamma' \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \quad \textit{dacă } \Gamma' = \Gamma[x \mapsto \tau]$$

$\vdash \lambda x.\lambda y.\lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t_x \to t$ dacă
$x \mapsto t_x \vdash \lambda y.\lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t$

Exemplu

Care este tipul expresiei următoare (dacă are)

$$\lambda x.\lambda y.\lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y$$

Aplicăm regula

$$(\text{:FN})\quad \frac{\Gamma' \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \quad \textit{dacă}\ \Gamma' = \Gamma[x \mapsto \tau]$$

$\vdash \lambda x.\lambda y.\lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t_x \to t$ dacă
$x \mapsto t_x \vdash \lambda y.\lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t$
Mai departe: $x \mapsto t_x \vdash \lambda y.\lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t_y \to t_0$ dacă
$x \mapsto t_x, y \mapsto t_y \vdash \lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t_0$ și, de mai sus,
$t = t_y \to t_0$

Exemplu

Care este tipul expresiei următoare (dacă are)

$$\lambda x.\lambda y.\lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y$$

Aplicăm regula

$$(\text{:FN})\quad \frac{\Gamma' \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \quad \textit{dacă}\ \Gamma' = \Gamma[x \mapsto \tau]$$

$\vdash \lambda x.\lambda y.\lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t_x \to t$ dacă
$x \mapsto t_x \vdash \lambda y.\lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t$
Mai departe: $x \mapsto t_x \vdash \lambda y.\lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t_y \to t_0$ dacă
$x \mapsto t_x, y \mapsto t_y \vdash \lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t_0$ și, de mai sus,
$t = t_y \to t_0$
Mai departe: $x \mapsto t_x, y \mapsto t_y \vdash \lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t_z \to t_1$
dacă $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t_1$ și, de mai
sus, $t_0 = t_z \to t_1$

Exemplu

Unde suntem

$\vdash \lambda x.\lambda y.\lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t_x \to t$ dacă
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t_1$ și $t_0 = t_z \to t_1$,
$t = t_y \to t_0$.

Aplicăm regula (:IF) $\dfrac{\Gamma \vdash e_b : bool \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e_b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t_1$ dacă
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y = 0 : \text{bool}$ și $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash z : t_1$ și
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash x/y : t_1$

Exemplu

Aplicăm regula

$$(:\text{COP})\quad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ o\ e_2 : bool} \quad \textit{dacă } o \in \{\leq, \geq, <, >, =\}$$

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y = 0$:`bool` dacă

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y$:`int` și $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash 0$:`int`

Aplicăm regula $(:\text{INT})\quad \Gamma \vdash n : int$ *dacă* n întreg

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash 0$:`int` este adevărat

Aplicăm regula

$$(:\text{IOP})\quad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ o\ e_2 : int} \quad \textit{dacă } o \in \{+, -, *, /\}$$

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash x/y$:`int` dacă

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash x$:`int` și $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y$:`int`

și, de mai sus, $t_1 =$`int`

Exemplu

Recapitulăm

$\vdash \lambda x.\lambda y.\lambda z.\ \mathtt{if}\ y = 0\ \mathtt{then}\ z\ \mathtt{else}\ x/y : t_x \to t$ dacă
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y : \mathtt{int}$ și $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash z : t_1$
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash x : \mathtt{int}$ și $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y : \mathtt{int}$
și $t_0 = t_z \to t_1$, $t = t_y \to t_0$, $t_1 = \mathtt{int}$.

Aplicăm regula (:VAR) $\Gamma \vdash x : \tau$ *dacă* $\Gamma(x) = \tau$

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y : t_y$ adevărat și, de mai sus $t_y = \mathtt{int}$
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash z : t_z$ adevărat și, de mai sus, $t_1 = t_z$
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash x : t_x$ adevărat și, de mai sus, $t_x = \mathtt{int}$

Exemplu

Finalizăm

$\vdash \lambda x.\lambda y.\lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : t_x \rightarrow t$ dacă
$t_0 = t_z \rightarrow t_1$, $t = t_y \rightarrow t_0$, $t_1 = \texttt{int}$, $t_y = \texttt{int}$, $t_1 = t_z$ și $t_x = \texttt{int}$.

Rezolvăm constrângerile și obținem

$\vdash \lambda x.\lambda y.\lambda z.\ \texttt{if}\ y = 0\ \texttt{then}\ z\ \texttt{else}\ x/y : \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int}$

Relația de asociere de tipuri în Prolog

Definim (recursiv) o relație de forma `type(Gamma, E, T)`, unde

- `Gamma` este o listă de perechi de forma `(X, T)` unde `X` este un identificator și `T` este o expresie de tip cu variabile
- `E` este o λ-expresie scrisă cu sintaxa descrisă mai sus
- `T` este o expresie de tip cu variabile

Sintaxa limbajului LAMBDA si sintaxa tipurilor

BNF LAMBDA

$$
\begin{array}{lll}
e & ::= & x \mid n \mid \mathit{true} \mid \mathit{false} \\
 & \mid & e + e \mid e < e \mid \mathtt{not}\ (e) \\
 & \mid & \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \\
 & \mid & \lambda x.e \mid e\ e \\
 & \mid & \mathtt{let}\ x = e\ \mathtt{in}\ e
\end{array}
$$

BNF TIPURI

$\tau ::=$ *int* [întregi]
$\mid$ *bool* [valori de adevăr]
$\mid$ $\tau \to \tau$ [funcții]
$\mid$ a [variabile de tip]

Sintaxa tipurilor

BNF

τ ::= *int* [întregi]
| *bool* [valori de adevăr]
| $\tau \rightarrow \tau$ [funcții]
| a [variabile de tip]

Verificarea sintaxei tipurilor în Prolog

```
is_type(X) :- var(X).         % X este variabila neinstantiata
is_type(int).                 % intregi
is_type(bool).                % valori de adevar
is_type(T1 -> T2) :-          % functii
   is_type(T1), is_type(T2).
```

Axiome

(:VAR) $\Gamma \vdash x : \tau \quad$ *dacă* $\Gamma(x) = \tau$

```
type(Gamma, X, T) :- atom(X), get(Gamma, X, T).
```

(:INT) $\Gamma \vdash n : int \quad$ *dacă n* întreg

```
type(_, I, int) :- integer(I).
```

(:BOOL) $\Gamma \vdash b : bool \quad$ *dacă* $b = true$ *or* $b = false$

```
type(_, true, bool).
type(_, false, bool).
```

Expresii

$$(\text{:IOP})\quad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ o\ e_2 : int} \quad \text{dacă } o \in \{+, -, *, /\}$$

```
type(Gamma, E1 + E2, int) :-
    type(Gamma, E1, int), type(Gamma, E2, int).
```

$$(\text{:COP})\quad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ o\ e_2 : bool} \quad \text{dacă } o \in \{\leq, \geq, <, >, =\}$$

```
type(Gamma, E1 < E2, bool) :-
    type(Gamma, E1, int), type(Gamma, E2, int).
```

$$(\text{:BOP})\quad \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1\ o\ e_2 : bool} \quad \text{dacă } o \in \{,\}$$

```
type(Gamma, and(E1, E2), bool) :-
    type(Gamma, E1, bool), type(Gamma, E2, bool).
```

Expresia condițională

$$(\text{:IF})\quad \frac{\Gamma \vdash e_b : bool \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e_b\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

```
type(Gamma, if(E, E1, E2), T) :-
    type(Gamma, E, bool),
    type(Gamma, E1, T),
    type(Gamma, E2, T).
```

Fragmentul funcțional

$$(:\textsc{fn})\quad \frac{\Gamma' \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \quad \textit{dacă } \Gamma' = \Gamma[x \mapsto \tau]$$

```
type(Gamma, X -> E, TX -> TE) :-
                         atom(X),
                         set(Gamma, X, TX, GammaX),
                         type(GammaX, E, TE).
```

$$(:\textsc{app})\quad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

```
type(Gamma, E1 $ E2, T) :-
                       type(Gamma, E, TE2 -> T),
                       type(Gamma, E2, TE2).
```

Tipurile variabile nu sunt suficiente

Tipurile variabile sunt destul de flexibile

- $\vdash \lambda x.x : t \to t$ pentru orice t
- $\vdash \texttt{if}\ (\lambda x.x)\ \mathit{true}\ \texttt{then}\ (\lambda x.x)\ 3\ \texttt{else}\ 4 : \texttt{int}$

Tipurile variabile nu sunt suficiente

Tipurile variabile sunt destul de flexibile

- $\vdash \lambda x.x : t \to t$ pentru orice t
- $\vdash \texttt{if}\ (\lambda x.x)\ \mathit{true}\ \texttt{then}\ (\lambda x.x)\ 3\ \texttt{else}\ 4 : \texttt{int}$

Dar tipul unei expresii este fixat:

$$\nvdash (\lambda id.\texttt{if}\ id\ \mathit{true}\ \texttt{then}\ id\ 3\ \texttt{else}\ 4)(\lambda x.x) : \texttt{int}$$

Tipurile variabile nu sunt suficiente

Tipurile variabile sunt destul de flexibile

- $\vdash \lambda x.x : t \to t$ pentru orice t
- $\vdash \texttt{if } (\lambda x.x)\ \mathit{true} \texttt{ then } (\lambda x.x)\ 3 \texttt{ else } 4 : \texttt{int}$

Dar tipul unei expresii este fixat:

$$\nvdash (\lambda id.\texttt{if } id\ \mathit{true} \texttt{ then } id\ 3 \texttt{ else } 4)(\lambda x.x) : \texttt{int}$$

Soluție

Pentru funcțiile cu nume, am vrea să fie ca și cum am calcula mereu tipul

$$\vdash \texttt{let } id = (\lambda x.x) \texttt{ in if } id\ \mathit{true} \texttt{ then } id\ 3 \texttt{ else } 4) : \texttt{int}$$

Operațional: redenumim variabilele de tip când instanțiem numele funcției

Scheme de tipuri

- Numim schemă de tipuri o expresie de forma $\langle\tau\rangle$, unde τ este e un expresie tip cu variabile
- variabilele dintr-o schemă nu pot fi constrânse e ca și cum ar fi cuantificate universal
- O schemă poate fi concretizată la un tip obișnuit substituindu-i fiecare variabilă cu orice tip (poate fi și variabilă)

Reguli pentru scheme

$$(\text{:LET})\quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma_1 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e2 : \tau} \quad \textit{dacă}\ \Gamma_1 = \Gamma[\langle\tau_1\rangle/x]$$

Reguli pentru scheme

$$(:\text{LET})\quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma_1 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e2 : \tau} \quad \textit{dacă}\ \Gamma_1 = \Gamma[\langle\tau_1\rangle/x]$$

```
type(Gamma, let(X, E1, E2), T) :-
    type(Gamma, E1, T1),
    copy_term(T1, FreshT1),   % redenumeste variabilele
                              % ca sa nu poata fi constranse
    set(Gamma, X, scheme(FreshT1), GammaX),
    type(GammaX, E2, T).
```

Reguli pentru scheme

(:SCH) $\Gamma \vdash x : \tau'$ *dacă* $\Gamma(x) = \langle\tau\rangle$ și $\tau' = \theta(\tau)$

```
type(Gamma, X, T) :-
    atom(X), get(Gamma, X, T), is_type(T).
type(Gamma, X, T) :-
    atom(X), get(Gamma, X, scheme(TX)),
    copy_term(T1, T).          % redenumeste variabilele
                               % ca sa poata fi constranse
```

Exemple

Prolog

```
  run(E) :-
    write("Program "),
    write(E),
    type(([],[]), E, T)
    ->  write(" has type "), write(T), nl
    ;   write(" doesn't type").

 ?- run(id -> if(id $ true, id $ 3, 4 )).
Program id->if(id$true,id$3,4) doesn't type
true.

?- run(let(id, x -> x, if(id $ true, id $ 3, 4 ))).
Program let(id,(x->x),if(id$true,id$3,4)) has type int
true
```

Succes la examen!