DD1361 Programmeringsparadigm HT16

Logikprogrammering 2

Dilian Gurov, TCS

Idag

Induktiva datatyper: Listor (inbyggd)

- Listor
- Strukturell induktion över listor
- Strängar

Läsmaterial

- Boken: kap. 4, 6
- PROLOG-fil: list.pl
- Handouts: Föreläsningsanteckningar

Listor

Induktiv definition

Listorna är en oändlig mängd av PROLOG-termer:

- En lista är antingen **tom** `[]`, eller en **konstruktion** `[a | l]` av ett element `a` ("huvud") och en lista `l` ("svans").
- Definition i *Backus–Naur Form* (BNF):

  `<Lst> ::= [] | [<El>|<Lst>]`

  där `<El>` är vilken som helst PROLOG-term.

Därmed matchar varje lista `l` antingen `[]` eller `[H | T]`, och vi kan använda detta för att ta isär (**destruera**) listor för att definiera predikat över listor med **strukturell induktion**.

Notationskonvention

Vi skriver `[a, b, c]` istället för `[a | [b | [c | []]]]`, och detta gör också PROLOGs "pretty-print" av listor.

Strukturell induktion över listor

För induktivt definierade datatyper kan vi användas av **strukturell induktion** för att definiera predikat över dem. Konkret, för listor:

- för tomma listan `[]`, definiera predikatet direkt;
- för sammansatta listan `[H | T]`, definiera predikatet med användning av samma predikat, beräknat över svansen `T`.

Om man följer principen garanterar man att predikatet blir "väldefinierad" över **alla** listor.

Läs också texten: *"The Priciple of Structural Induction"* !

Längden på en lista

Längden på en lista `L` är antalet element `N` i listan.

Definition med strukturell induktion:

Längden på en lista

Längden på en lista `L` är antalet element `N` i listan.

Definition med strukturell induktion:

- längden på tomma listan `[]`

Längden på en lista

Längden på en lista `L` är antalet element `N` i listan.

Definition med strukturell induktion:

- längden på tomma listan `[]`
  är `0`;

Längden på en lista

Längden på en lista `L` är antalet element `N` i listan.

Definition med strukturell induktion:

- längden på tomma listan `[]`
  är `0`;
- längden på sammansatta listan `[H | T]`

Längden på en lista

Längden på en lista `L` är antalet element `N` i listan.

Definition med strukturell induktion:

- längden på tomma listan `[]`
  är `0`;
- längden på sammansatta listan `[H | T]`
  är längden på svansen `T` plus `1`.

`listLength(L, N)`

I Prolog:

```
listLength([], 0).
listLength([_ | T], N) :-
  listLength(T, NT),
  N is NT + 1.
```

Notera att vi använder operatorn "`is`" istället för "=". Varför?

Notera också hur vi använder mönster-matchning i första argumentet för att åstadkomma datatyp-destruktionen som är nödvändig för strukturella induktionen.

Kan vi vända på ordningen på de två konjunkterna?

Finns även inbyggd som `length(L, N)`.

Kontrollflödet vid `listLength([a, b], N).`

Fråga: `listLength([a, b], N).`

Kontrollflödet vid `listLength([a, b], N).`

Fråga: `listLength([a, b], N).`

- skapar instans av andra regeln:
  `listLength([A1 | T1], N1) :- listLength(T1, NT1),`
  `N1 is NT1 + 1.`
  unifierar `A1=a, T1=[b], N1=N`

Kontrollflödet vid `listLength([a, b], N).`

Fråga: `listLength([a, b], N).`

- skapar instans av andra regeln:
  `listLength([A1 | T1], N1) :- listLength(T1, NT1),`
  `N1 is NT1 + 1.`
  unifierar `A1=a, T1=[b], N1=N`
- `listLength([b], NT1).` (rekursivt anrop)

Kontrollflödet vid `listLength([a, b], N).`

Fråga: `listLength([a, b], N).`

- skapar instans av andra regeln:
  `listLength([A1 | T1], N1) :- listLength(T1, NT1), N1 is NT1 + 1.`
  unifierar `A1=a, T1=[b], N1=N`
- `listLength([b], NT1).` (rekursivt anrop)
  - skapar instans av andra regeln:
    `listLength([A2 | T2], N2) :- listLength(T2, NT2), N2 is NT2 + 1.`
    unifierar `A2=b, T2=[], N2=NT1`

Kontrollflödet vid `listLength([a, b], N).`

Fråga: `listLength([a, b], N).`

- skapar instans av andra regeln:
  `listLength([A1 | T1], N1) :- listLength(T1, NT1), N1 is NT1 + 1.`
  unifierar `A1=a, T1=[b], N1=N`
- `listLength([b], NT1).` (rekursivt anrop)
  - skapar instans av andra regeln:
    `listLength([A2 | T2], N2) :- listLength(T2, NT2), N2 is NT2 + 1.`
    unifierar `A2=b, T2=[], N2=NT1`
  - `listLength([], NT2).` (rekursivt anrop)

Kontrollflödet vid `listLength([a, b], N).`

Fråga: `listLength([a, b], N).`

- skapar instans av andra regeln:
  `listLength([A1 | T1], N1) :- listLength(T1, NT1), N1 is NT1 + 1.`
  unifierar `A1=a, T1=[b], N1=N`
- `listLength([b], NT1).` (rekursivt anrop)
  - skapar instans av andra regeln:
    `listLength([A2 | T2], N2) :- listLength(T2, NT2), N2 is NT2 + 1.`
    unifierar `A2=b, T2=[], N2=NT1`
  - `listLength([], NT2).` (rekursivt anrop)
    - unifierar mot första regeln, `NT2=0`

Kontrollflödet vid `listLength([a, b], N).`

Fråga: `listLength([a, b], N).`

- skapar instans av andra regeln:
  `listLength([A1 | T1], N1) :- listLength(T1, NT1), N1 is NT1 + 1.`
  unifierar `A1=a, T1=[b], N1=N`
- `listLength([b], NT1).` (rekursivt anrop)
  - skapar instans av andra regeln:
    `listLength([A2 | T2], N2) :- listLength(T2, NT2), N2 is NT2 + 1.`
    unifierar `A2=b, T2=[], N2=NT1`
  - `listLength([], NT2).` (rekursivt anrop)
    - unifierar mot första regeln, `NT2=0`
  - `NT1 is 0 + 1.`

Kontrollflödet vid `listLength([a, b], N).`

Fråga: `listLength([a, b], N).`

- skapar instans av andra regeln:
  `listLength([A1 | T1], N1) :- listLength(T1, NT1), N1 is NT1 + 1.`
  unifierar `A1=a, T1=[b], N1=N`
- `listLength([b], NT1).` (rekursivt anrop)
  - skapar instans av andra regeln:
    `listLength([A2 | T2], N2) :- listLength(T2, NT2), N2 is NT2 + 1.`
    unifierar `A2=b, T2=[], N2=NT1`
  - `listLength([], NT2).` (rekursivt anrop)
    - unifierar mot första regeln, `NT2=0`
  - `NT1 is 0 + 1.`
    - evaluerar `0+1` till `1`, unifierar `NT1=1`

Kontrollflödet vid `listLength([a, b], N).`

Fråga: `listLength([a, b], N).`

- skapar instans av andra regeln:
  `listLength([A1 | T1], N1) :- listLength(T1, NT1), N1 is NT1 + 1.`
  unifierar `A1=a, T1=[b], N1=N`
- `listLength([b], NT1).` (rekursivt anrop)
  - skapar instans av andra regeln:
    `listLength([A2 | T2], N2) :- listLength(T2, NT2), N2 is NT2 + 1.`
    unifierar `A2=b, T2=[], N2=NT1`
  - `listLength([], NT2).` (rekursivt anrop)
    - unifierar mot första regeln, `NT2=0`
  - `NT1 is 0 + 1.`
    - evaluerar `0+1` till `1`, unifierar `NT1=1`
- `N is 1 + 1.`

Kontrollflödet vid `listLength([a, b], N).`

Fråga: `listLength([a, b], N).`

- skapar instans av andra regeln:
  `listLength([A1 | T1], N1) :- listLength(T1, NT1), N1 is NT1 + 1.`
  unifierar `A1=a, T1=[b], N1=N`
- `listLength([b], NT1).` (rekursivt anrop)
  - skapar instans av andra regeln:
    `listLength([A2 | T2], N2) :- listLength(T2, NT2), N2 is NT2 + 1.`
    unifierar `A2=b, T2=[], N2=NT1`
  - `listLength([], NT2).` (rekursivt anrop)
    - unifierar mot första regeln, `NT2=0`
  - `NT1 is 0 + 1.`
    - evaluerar `0+1` till `1`, unifierar `NT1=1`
- `N is 1 + 1.`
  - evaluerar `1+1` till `2`, unifierar `N=2`

Kontrollflödet vid `listLength([a, b], N).`

Fråga: `listLength([a, b], N).`

- skapar instans av andra regeln:
  `listLength([A1 | T1], N1) :- listLength(T1, NT1), N1 is NT1 + 1.`
  unifierar `A1=a, T1=[b], N1=N`
- `listLength([b], NT1).` (rekursivt anrop)
  - skapar instans av andra regeln:
    `listLength([A2 | T2], N2) :- listLength(T2, NT2), N2 is NT2 + 1.`
    unifierar `A2=b, T2=[], N2=NT1`
  - `listLength([], NT2).` (rekursivt anrop)
    - unifierar mot första regeln, `NT2=0`
  - `NT1 is 0 + 1.`
    - evaluerar `0+1` till `1`, unifierar `NT1=1`
- `N is 1 + 1.`
  - evaluerar `1+1` till `2`, unifierar `N=2`

Svar: `N=2`

Kontrollflödet vid `listLength(L, 2)`.

Från KS:en HT15:

```
Fråga: length(L, 2).
- misslyckas med första regeln, därför att 0 och 2 inte
  kan unifieras;
- skapar instans av andra regeln:
  length([A1|T1], N1) :- length(T1, NT1), N1 is NT1+1.
  unifierar: L=[A1|T1], N1=2;
  -- length(T1, NT1).
     --- lyckas med första regeln;
         unifierar: T1=[], NT1=0;
  -- 2 is 0+1.
     --- evaluerar 0+1 till 1;
     --- misslyckas, därför att 2 och 1 inte kan unifieras;
     --- backtrackar;
  -- length(T1, NT1).
     --- skapar instans av andra regeln:
         length([A2|T2], N2) :- length(T2, NT2), N2 is NT2+1.
         unifierar: T1=[A2|T2], N2=NT1;
         ---- length(T2, NT2).
              ----- lyckas med första regeln;
                    unifierar: T2=[], NT2=0;
         ---- NT1 is 0+1.
              ----- evaluerar 0+1 till 1;
              ----- unifierar: NT1=1;
  -- 2 is 1+1.
     --- evaluerar 1+1 till 2;
     --- lyckas, därför att 2 kan unifieras med 2.
Svar: L=[A1|[A2|[]]] som presenteras som L=[A1, A2].
```

Medlemskap i en lista: `in(X, L)`

Medlemstest som ska vara sant om och bara om `X` finns i listan `L`.

Medlemskap i en lista: `in(X, L)`

Medlemstest som ska vara sant om och bara om `X` finns i listan `L`.

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

Strukturella induktionen är över listan `L` (dvs andra argumentet).
`in(X, [])` är alltid falskt, därför ingen regel för tom lista!

Finns även inbyggd som `member(X, L)`.

Kontrollflödet vid `in(2, L), in(1, L).`

Fråga: `in(2, L), in(1, L).`

Kontrollflödet vid `in(2, L), in(1, L).`

Fråga: `in(2, L), in(1, L).`

- `in(2, L).`

Kontrollflödet vid `in(2, L), in(1, L).`

Fråga: `in(2, L), in(1, L).`

- `in(2, L).`
  - skapar instans av första regeln: `in(H1, [H1 | A1]).`
    unifierar `H1=2, L=[2 | A1]`

Kontrollflödet vid `in(2, L), in(1, L).`

Fråga: `in(2, L), in(1, L).`

- `in(2, L).`
  - skapar instans av första regeln: `in(H1, [H1 | A1]).`
    unifierar `H1=2, L=[2 | A1]`
- `in(1, [2 | A1]).`

Kontrollflödet vid `in(2, L), in(1, L).`

Fråga: `in(2, L), in(1, L).`

- `in(2, L).`
  - skapar instans av första regeln: `in(H1, [H1 | A1]).` unifierar `H1=2, L=[2 | A1]`
- `in(1, [2 | A1]).`
  - skapar instans av första regeln: `in(H2, [H2 | A2]).` misslyckas med unifieringen (varför?)

Kontrollflödet vid `in(2, L), in(1, L).`

Fråga: `in(2, L), in(1, L).`

- `in(2, L).`
  - skapar instans av första regeln: `in(H1, [H1 | A1]).`
    unifierar `H1=2, L=[2 | A1]`
- `in(1, [2 | A1]).`
  - skapar instans av första regeln: `in(H2, [H2 | A2]).`
    misslyckas med unifieringen (varför?)
  - skapar instans av andra regeln:
    `in(X1, [A3 | T1]) :- in(X1, T1).`
    unifierar `X1=1, A3=2, T1=A1`

Kontrollflödet vid `in(2, L), in(1, L).`

Fråga: `in(2, L), in(1, L).`

- `in(2, L).`
  - skapar instans av första regeln: `in(H1, [H1 | A1]).`
    unifierar `H1=2, L=[2 | A1]`
- `in(1, [2 | A1]).`
  - skapar instans av första regeln: `in(H2, [H2 | A2]).`
    misslyckas med unifieringen (varför?)
  - skapar instans av andra regeln:
    `in(X1, [A3 | T1]) :- in(X1, T1).`
    unifierar `X1=1, A3=2, T1=A1`
    - `in(1, A1).`

Kontrollflödet vid `in(2, L), in(1, L).`

Fråga: `in(2, L), in(1, L).`

- `in(2, L).`
  - skapar instans av första regeln: `in(H1, [H1 | A1]).`
    unifierar `H1=2, L=[2 | A1]`
- `in(1, [2 | A1]).`
  - skapar instans av första regeln: `in(H2, [H2 | A2]).`
    misslyckas med unifieringen (varför?)
  - skapar instans av andra regeln:
    `in(X1, [A3 | T1]) :- in(X1, T1).`
    unifierar `X1=1, A3=2, T1=A1`
    - `in(1, A1).`
    - skapar instans av första regeln: `in(H3, [H3 | A4]).`
      unifierar `H3=1, A1=[1 | A4]`

Kontrollflödet vid `in(2, L), in(1, L).`

Fråga: `in(2, L), in(1, L).`

- `in(2, L).`
  - skapar instans av första regeln: `in(H1, [H1 | A1]).`
    unifierar `H1=2, L=[2 | A1]`
- `in(1, [2 | A1]).`
  - skapar instans av första regeln: `in(H2, [H2 | A2]).`
    misslyckas med unifieringen (varför?)
  - skapar instans av andra regeln:
    `in(X1, [A3 | T1]) :- in(X1, T1).`
    unifierar `X1=1, A3=2, T1=A1`
    - `in(1, A1).`
    - skapar instans av första regeln: `in(H3, [H3 | A4]).`
      unifierar `H3=1, A1=[1 | A4]`

Svar: `L =[2, 1 | A4]`

Listkonkatenering: `concatenate(X, Y, Z)`

Ska vara sant om `Z` är konkateneringen av listan `X` med listan `Y`.

Listkonkatenering: `concatenate(X, Y, Z)`

Ska vara sant om `Z` är konkateneringen av listan `X` med listan `Y`.

```
concatenate([], Y, Y).
concatenate([HX | TX], Y, [HX | TZ]) :-
  concatenate(TX, Y, TZ).
```

Strukturella induktionen är över listan `X` (dvs första argumentet).

Finns även inbyggd som `append(X, Y, Z)`.

Lägg till ett element: `appendEl(X, L, NL)`

Ska vara sant om listan `NL` är listan `L` med med elementet `X` lagt till i slutet.

Lägg till ett element: `appendEl(X, L, NL)`

Ska vara sant om listan `NL` är listan `L` med med elementet `X` lagt till i slutet.

```
appendEl(X, [], [X]).
appendEl(X, [H | T], [H | Y]) :-
  appendEl(X, T, Y).
```

Strukturella induktionen är över listan `L` (dvs andra argumentet).

Listomvändning: `rev(X, Y)`

Ska vara sant om `Y` är omvända listan `X`.

Listomvändning: `rev(X, Y)`

Ska vara sant om `Y` är omvända listan `X`.

```
rev([], []).
rev([H | T], X) :-
  rev(T, RT),
  appendEl(H, RT, X).
```

Finns även inbyggd som `reverse(X, Y)`.

Strängar

Strängar

- Är symbolsekvenser.
- Representeras i PROLOG internt som listor av heltal. Varje tal representerar därmed en symbol (ASCII-koden).
- Inbyggda predikatet `atom_codes(X, Y)` är sant när `Y` är strängen (dvs heltalslistan) som motsvarar atomen `X`.

Listsortering

Det finns många sätt att sortera listor.

Här ska vi implementera **permutationssortering**, som illustration av hur vi kan använda backtracking som ett styrka för att realisera en programmeringsteknik som kallas för **generera och testa**.

Generera och testa

Denna programmeringsteknik använder sig av backtrackningen för att successivt generera en möjlig kandidatlösning, testa om den uppfyller villkoren för att vara en korrekt lösning, sedan backtracka och generera en möjlig kandidatlösning till, testa den, osv.

Generella strukturen ser ut så här:

```
problem(Problem, Solution) :-
  generate(Problem, Solution),
  test(Solution).
```

Tekniken är särskilt lämplig när algoritmiska komplexiteten av problemet som ska lösas är högt.

Listsortering: `permSort(+X, ?Y)`

Vi utgår från matematiska definitionen av sortering: att sortera en lista kan definieras som att skapa (dvs beräkna) en **sorterad permutation** av ursprungliga listan:

```
permSort(X, Y) :-
  permutation(X, Y),     \\ generera permutation
  sorted(Y).             \\ testa om sorterad
```

Programmet använder (inbyggda) predikatet `permutation(X, Y)` för att generera en permutation av listan `X`, som sedan testas med predikatet `sorted(Y)` om den är sorterad eller inte.

Del 1: `permutation(+X, ?Y)`

Ska vara sant om `Y` är en permutation av `X`.

Strukturella induktionen är på första listan `X`.

Vi utgår från följande logiska karakteriseringen av permutation:
`Y` är en permutation av `X` om huvudet på `X` finns i `Y`, och om tagit bort från `Y`, resulterade listan är en permutation av svansen på `X`.

Del 1: `permutation(+X, ?Y)`

Ska vara sant om `Y` är en permutation av `X`.

Strukturella induktionen är på första listan `X`.

Vi utgår från följande logiska karakteriseringen av permutation:
`Y` är en permutation av `X` om huvudet på `X` finns i `Y`, och om tagit bort från `Y`, resulterade listan är en permutation av svansen på `X`.

```
permutation([], []).
permutation([E | X], Y) :-
  permutation(X, Y1),
  append(Y2, Y3, Y1),
  append(Y2, [E | Y3], Y).
```

Del 2: `sorted(X)`

Definieras här bara för heltalslistor!

Del 2: `sorted(X)`

Definieras här bara för heltalslistor!

```
sorted([]).
sorted([X]).
sorted([X, Y | L]) :-
  X =< Y,
  sorted([Y | L]).
```