

Logikprogrammering

Fakta
Regler
Forespørgsler
Eksempler
Grafer og grafrelationer
Specifikation af de naturlige tal
Datastrukturer
Lister
Unification

PS3 - Logikprogrammering

© Kurt Nørmark, Aalborg Universitet

9/12/96

s. 1

Noter

Til dette kapitel læser vi kapitel 5 i *Programming Language Essentials* af Bal og Grune fra Addison Wesley. Endvidere supplerer vi med *Prolog a tutorial Introduction* af Lu og Mead (fra WWW).

Som baggrundsmateriale kan endvidere anbefales *Programming in Prolog* af Clocksin og Mellish fra Springer Verlag.

Det logiske programmeringsparadigme.

- Inspireret af *automatisk bevisførelse* inden for det datalogiske område, som kaldes "kunstig intelligens" (AI).
- Baseret på
 - en mængde af fakta
 - en mængde af *regler*, som er regler hvorom ny viden kan afledes fra eksisterende viden.
 - *forespørgsler*.
- *Deklarativ*: Et program manifesterer sig som en mængde erklæringer af fakta og sammenhænge (regler).
- Underliggende besvares en forespørgsel ved *systematisk søgning* i aksiomerne og inferensreglerne.

- Fremmedartet for programmører, som ikke er trænet i paradigmet.
- Fremtræder som grundlæggende forskelligt fra de andre paradigmer.
- Først og fremmest nyttigt når et problem kan formuleres som et spørgsmål om, og under hvilke omstændigheder, et udsagn er korrekt.

PS3 - Logikprogrammering

Noter

Denne slide opsummerer vores umiddelbare paradigmekarakteristik fra introduktionsforelæsningsen.

Fakta

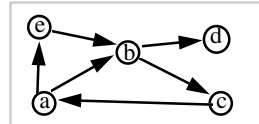
- Et *faktum* udtrykker en sammenhæng, som vi ved er opfyldt.
- Alle ikke-udtrykte fakta er ikke opfyldt.
- Formen (syntaksen) af et faktum:

```
relation(const1, ..., constn).
```

↙ *prædikat*

- Betydningen (semantikken) af et faktum:
 - const₁, ..., const_n står i relation til hinanden på en bestemt måde.
 - tuplen (const₁, ..., const_n) tilhører relation.
- Leksikalske konventioner i Prolog
 - prædikater starter med en lille bogstav
 - konstanter starter med et lille bogstav.

```
edge(a, b).    edge(b, c).  
edge(a, e).    edge(c, a).  
edge(b, d).    edge(e, b).
```



PS3 - Logikprogrammering

Noter

Her og i det følgende lægger vi os tæt op ad sproget Prolog, som er det klart dominerende sprog i det logiske programmeringsparadigme.

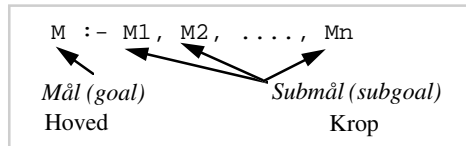
Som vi vil se lidt senere kan man også have fakta, hvori der indgår variable. Sådanne fakta skal holde for alle mulige værdier af variablene.

I fakta kan vi udtrykke allehånde nyttige sammenhænge - ligesom det kendes fra relationer i relationsdatabaser. Og som vi vil se senere, er vi i stand til at søge på elementerne i relationerne i forbindelse med besvarelse af forespørgsler.

Eksemplet er taget fra *Prolog a tutorial introduction*.

Regler (1).

- En *regel* udtrykker en sammenhæng, som vi kan slutte os til ud fra andre sammenhænge.
- Formen af en regel:



- Syntaks:
 - mål ::= prædikat($term_1, \dots, term_n$).
 - term ::= konstant | variable | funktor($term_1, \dots, term_m$)
- Betydningen (semantikken) af en regel:
 - Hvis **M1 og M2 og ... Mn** er opfyldt så er M opfyldt.
 - Tankegang: M er opfyldt såfremt vi kan vise at både M1, M2, ..., Mn er opfyldt, hvilket vi derfor prøver at bevise.
 - Bemærk enkeltimplikation: M kan evt. opfyldes på andre måder

PS3 - Logikprogrammering

Noter

Bemærk, at der ikke er syntaktisk forskel mellem

- et mål, udtrykt som et prædikat på et antal termer og
- en funktor på et antal termer.

En funktor tillader os at lave datastrukturer, som aggregerer et antal termer. Bemærk at funktorer kan nestes inden i hinanden, i modsætning til prædikater, som hører til "foruden".

Regler (2)

```
on_two_edge(Node1,Node2) :-  
    edge(Node1,SomeNode), edge(SomeNode,Node2).  
path(Node1,Node2) :- edge(Node1,Node2).  
path(Node1,Node2) :-  
    edge(Node1,SomeNode), path(SomeNode,Node2).
```

- Leksikalsk konvention:
 - Variable har stort begyndelsesbogstav.
- Semantisk konvention:
 - Alle variable i en regel er universel kvantificeret.
- Fakta i forhold til regler:
 - Et faktum kan opfattes som en regel, hvor kroppen er tom.
- **And** og **or** i regler:
 - **And** kan udtrykkes umiddelbart i reglen gennem konjunktionen af delmål.
 - **Or** kan udtrykkes ved flere regler med samme hoved.

PS3 - Logikprogrammering

Noter

Eksemplet foroven bygger videre på den graf, som vi erklærede nogle få sider tilbage.

Prædikatet `on_two_edge(N1,N2)` udtrykker at `N1` og `N2` er placeret på en "to-kant", som udspringer i `N1` og ender i `N2`. Der er altså nøjagtig én knude imellem `N1` og `N2` på denne kant.

Prædikatet `path(N1,N2)` udtrykker, at der findes en stil fra `N1` til `N2` af længde én eller mere.

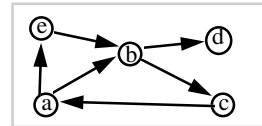
Spørgsmål.

- Et *spørgsmål (query)* udtrykker et start-mål (initial goal), som ønskes bekræftet (“Ja”) eller afkræftet (“Nej”) i forhold til fakta og regler. Videre ønskes besvaret under hvilke omstændigheder målet kan opfyldes.
- Formen af et spørgsmål:
 - Konjunktion af af del-mål: $?- M1, M2, \dots, Mn.$
- Eventuelle variable i et spørgsmål er eksistentielt kvantificerede:
 - Eksisterer der værdier af variablene således at målet kan opfyldes?
 - I så fald: hvilke?

```
1 ?- on_two_edge(a,d).
Yes

2 ?- on_two_edge(X,b).
X = a ;
X = c ;
No

3 ?- path(d,a).
No
```



PS3 - Logikprogrammering

Noter

Symbolet $?-$ er “prompten” i mange Prolog systemer, som indikerer at der kan stilles et spørgsmål.

I interaktion nummer 2 i eksemplet ser vi et semikolon efter $X = a$ og $X = b$. Disse tegn er tastet af programmøren, som har stillet spørgsmålet. Med semikolon beder programmøren om at få en evt. anden mulig binding af variabelen X . Når man taster semikolon beder man med andre ord søgeprocessen om at “backtrack”. Bindingen af X til a brydes, og vi forsøger at finde en anden binding, som opfylder spørgsmålet.

Eksempel: Fakta og regler om naturlige tal (1).

```
is_zero(null).  
  
leq(null,J).  
leq(succ(I),succ(J)) :- leq(I,J).  
  
add(null,J,J).  
add(succ(I),J,succ(R)) :-  
  add(I,J,R).  
  
mult(null,J,null).  
mult(J,null,null).  
mult(succ(I),J,R) :-  
  mult(I,J,R1), add(J,R1,R).
```

- Strukturer og funktorer.
 - $\text{succ}(X)$ er et eksempel på en *struktur*.
 - succ kaldes en *funktor*, X en komponent.
 - Generelt set kan en struktur opfattes som en *datastruktur*.

```
type Nat0  
functions  
  null: -> Nat0  
  succ: Nat0 -> Nat0  
  is-zero: Nat0 -> Boolean  
  leq: Nat0 x Nat0 -> Boolean  
  add: Nat0 x Nat0 -> Nat0  
  mult: Nat0 x Nat0 -> Nat0  
axioms  
for all i, j in Nat0  
  is-zero(null) = true  
  is-zero(succ(i)) = false  
  leq(null(), j) = true  
  leq(succ(i), null()) = false  
  leq(succ(i), succ(j)) = leq(i, j)  
  add(null(), j) = j  
  add(succ(i), j) = succ(add(i, j))  
  mult(null(), j) = null().  
  mult(j, null()) = null().  
  mult(succ(i), j) = add(mult(i, j), j)
```

PS3 - Logikprogrammering

Noter

I dette eksempel tager vi udgangspunkt i en *algebraisk specifikation* af de naturlige tal fra og med 0. I denne specifikation udtrykkes at vi har null og $\text{succ}(N)$ som konstruktører af de naturlige tal. Som eksempler på operatorer på de naturlige tal ser vi på add og mult (addition hhv. multiplikation). Vi ser også på prædikatet leq , der returnerer hvorvidt et tal er mindre eller lig med et andet.

Værdier af den abstrakte datatype, i dette eksempel, naturlige tal, udtrykkes i termer af konstruktørerne. Et vilkårligt naturligt tal kan udtrykkes som $\text{succ}(\text{succ}(\dots \text{null}() \dots))$; altså som det antal gange vi skal addere tallet 1 til 0 for at opnå det pågældende tal.

Man kan sige, at et objekt repræsenteres ved dets *operationshistorie*.

Reduktion af udtryk: Givet et vilkårligt udtryk U kan man omskrive U ved brug af en eller flere af ligningerne. En særlig interessant omskrivning består i at reducere udtrykket til et udtryk, som kun består af konstruktører.

Grundlaget for specifikationen af Nat0 er Peano's klassiske definition af de naturlige tal.

Til venstre viser vi en Prolog formulering af den algebraiske specifikation. Bemærk, at succ i Prolog programmet er en funktor, og ikke et prædikat.

leq : Vi har ikke en regel, der udtrykker, at det er falsk at et stort tal er mindre eller lig med et lille tal. Når vi ikke har erklæret noget for sandt i Prolog, er det implicit falsk! Dermed er der altså ikke en én til én korrespondance mellem ligninger og regler.

add : Bemærk, at vi har brug for tre parametre til add , og ikke to som i specifikationen til højre. Dette skyldes at vi har brug for at udtrykke en relation mellem de to addender og resultatet. add er et prædikat i Prolog programmet, og en funktion i den algebraiske specifikation.

Eksempel: Fakta og regler om naturlige tal (2).

```
1 ?- is_zero(succ(null)).
No

2 ?- add(succ(null), succ(succ(null)), X).
X = succ(succ(succ(null))) ;
Yes

3 ?- add(succ(null), succ(null), succ(succ(null)) ).
Yes

4 ?- mult(succ(succ(null)), succ(succ(succ(null))), X).
X = succ(succ(succ(succ(succ(succ(null))))))
Yes

5 ?- mult(succ(succ(null)), Y,
          succ(succ(succ(succ(succ(succ(null)))))) ).
Y = succ(succ(succ(null)))
Yes
```

PS3 - Logikprogrammering

Noter

I interaktion 1 spørger vi om, om 1 er 0. Dette viser sig ikke at være tilfældet.

I interaktion 2 spørger vi om resultatet af $1 + 2$. Resultatet ses at være 3.

I interaktion 3 får vi bekræftet at $1+1$ er lig med 2

I interaktion 4 bliver vi lidt mere avancerede, idet vi beregner $2 * 3$. Svaret er, som det ses, 6.

I interation 5 spørger vi om der findes et naturligt tal Y, hvorom der gælder at $2 * Y = 6$. Svaret er, som forventet, 3. Bemærk Prolog's styrker her, idet vi jo i relaliteten har både multiplikation og division i reglerne mult.

I ovenstående er der måske en usikkerhed om semikolons og svarene "Yes" og "No2. Prøv det selv af.

Hvordan besvares et on_two_edge spørgsmål (1) ?

```
?- on_two_edge(a,d).  
Yes
```

```
edge(a,b).      edge(b,c).  
edge(a,e).      edge(c,a).  
edge(b,d).      edge(e,b).  
  
on_two_edge(Node1,Node2) :-  
    edge(Node1,SomeNode),  
    edge(SomeNode,Node2).
```

- Det initiale mål erstattes af reglens krop:
 - Node1 = a, Node2 = d.
- Bevis: `edge(a, SomeNode), edge(SomeNode, d)`.
- Ved søgning gennem facts og regler forsøger vi at vise at der findes en knude `SomeNode`, så `edge(a, SomeNode)` er opfyldt. `SomeNode = b` er en sådan knude.
- Vi forsøger at bevise at `edge(b, d)`. Dette er imidlertid let, da det er et faktum.
- QED.

PS3 - Logikprogrammering

Noter

På denne slide kigger vi på detaljerne i den bevisførelse, som Prolog udfører, når den besvarer spørgsmål. Spørgsmålet er i den fede ramme. I rammen øverst til højre ser vi de relevante regler.

Hvordan besvares et `on_two_edge` spørgsmål (2) ?

```
?- on_two_edge(X,b).  
X = a ;  
X = c ;  
No
```

```
edge(a,b).      edge(b,c).  
edge(a,e).      edge(c,a).  
edge(b,d).      edge(e,b).  
  
on_two_edge(Node1,Node2) :-  
    edge(Node1,SomeNode),  
    edge(SomeNode,Node2).
```

- Det initiale mål erstattes af reglens krop: `Node1 = X, Node2 = b`.
- Bevis: `edge(X, SomeNode), edge(SomeNode, b)`.
- Vi afprøver om første kant (a,b) opfylder ovenstående: Nej.
- Vi afprøver nu, om anden kant (a,e) opfylder ovenstående: Ja.
- `X = a` er et muligt svar.
- Vi taster “;”: Vi ønsker vi finde evt. andre X-er.
- Vi backtracker fra punkt 4: (b,d), (b,c): Ingen success.
- Vi prøver kanten (c,a), altså `X = c`. `SomeNode` bliver a. Da der findes en kant (a,b) er `X=c` også en løsning.
- Vi taster “;”: Vi ønsker vi finde evt. andre X-er.
- Den sidste kant (e,b) giver ikke anledning til en løsning. Derfor ‘no’ tilsidst.

PS3 - Logikprogrammering

Noter

Hvordan besvares et add spørgsmål?

```
add(null,J,J).  
add(succ(I),J,succ(R)) :-  
    add(I,J,R).
```

```
?- add(succ(null), succ(succ(null)), X).  
X = succ(succ(succ(null))) ;  
Yes
```

- Spørgsmålet matcher ikke den første regel, men derimod den anden.
- Det initiale mål erstattes af den anden regel's krop med
 - $I = \text{null}$, $J = \text{succ}(\text{succ}(\text{null}))$ og $X = \text{succ}(R)$.
- Bevise: $\text{add}(\text{null}, \text{succ}(\text{succ}(\text{null})), R)$.
- Spørgsmålet matcher nu den første regel med
 - $R = \text{succ}(\text{succ}(\text{null}))$.
- I det oprindelige spørgsmål er
 - $X = \text{succ}(R) = \text{succ}(\text{succ}(\text{succ}(\text{null})))$.

PS3 - Logikprogrammering

Noter

Unification.

- *Unification* er en form for *generaliseret pattern-matching* mellem to termer.
 - $\text{Unify}(\text{Term1}, \text{Term2}) \rightarrow \text{Substitution}$.
- Intuitivt er en *substitution* en mængde af formen:
 - $\{V1 = t1, V2 = t2, \dots, Vm = tm\}$
 - V_i er variable og t_i er termer.
- Intuitivt er substitutionen den mængde af variabelbindinger, som gør Term1 og Term2 syntaktisk identiske.
 - Vi anvender en substitution på en term ved simultant at erstatte alle variable med de tilhørende termer.

Noter

Unification er en helt central teknik i Prolog's bevisførelse. Vi har allerede gennemgået et antal eksempler, hvor unification har været i sving. Her ser vi lidt nærmere på begrebet.

Vi erindrer om, at en term er enten en konstant, en variable, eller en funktor af et antal termer.

I Prolog a tutorial introduction gennemgås unification i rimelig detalje - og på en mere formaliseret måde end det foregår her.

En iterativ unification algoritme.

```
Function Unify(pred1(termlist1), pred2(termlist2): Atom): Substitution;  
let result be empty  
      failure be false  
in  
  if (pred1 = pred2) and (length(termlist1) = length(termlist2))  
  then push all pairs from termlist1 and termlist2 onto termstack  
        while not (empty(termstack)) and not fail  
        do pop (X,Y) from termstack  
          case  
            X is a variable that does not occur in Y:  
              substitute Y for X in the stack;  
              add X=Y to the result  
            Y is a variable that does not occur in X:  
              substitute X for Y in the stack;  
              add Y = X to the result  
            X and Y are identical constants or variables:  
              continue  
            X = f(x1, ..., xn) og Y = f(y1, ... , yn)  
              push (xi, yi) på stakken for i = 1, ..., n.  
          otherwise  
            Failure := true  
        end --case  
  else result = failure
```

PS3 - Logikprogrammering

Noter

Denne algoritme stammer essentielt set fra Prolog a tutorial introduction. Jeg har lavet nogle mindre omskrivninger og præciseringer.

Parametrene til unify ser underlige ud. Det er fordi vi har foldet prædikatet og dets termer ud i selve parameterlisten. Alternativt kunne vi blot have overført to "atomer" atom1 og atom2, og vente med opsplitning i prædikat og termliste til let konstruktionen.

Bemærk navnet "atom", som undertiden bruges for et prædikat efterfulgt af en tupel af termer.

Lister i Prolog.

- En liste er enten
 - tom []
 - eller på formen [element1 | hale]
- Notation:
 - [a, b, c, d]
- Basal konstruktion:
 - .(a, .(b, .(c, .(d, []))))
- Selektion: via unification

```
my_append([],List,List).
my_append([H|Tail], X, [H|Newtail]) :-
    my_append(Tail, X, Newtail).

1 ?- my_append([a,b,c], Y, [a,b,c,d,e]).
Y = [d,e]
Yes

2 ?- my_append([a,b,c], [d], Z).
Z = [a,b,c,d]
Yes
```

PS3 - Logikprogrammering

Noter

Lister i Prolog er ganske nøje modelleret efter listebegrebet i Lisp, som vi vender tilbage til ved en senere forelæsning på dette kursus.

Operatoren “.” svarer til cons i Lisp.

[a, b, c, d] svarer til (a b c d) i Lisp.

(a, .(b, .(c, .(d, [])))) svarer til Lisp dot-notationen (a . (b . (c . (d. ())))). Begge er lige skrækkelige at se på.

I Prolog har vi ikke behov for car og cdr, som de kendes fra Lisp. Patternmatching og unification løser destrukturen af en liste i hoved og hale for os.

Append relationen er en klassiker, som næsten alle Prolog bøger og introduktioner har med. Vi gør ingen undtagelse her.

Hvordan besvares et append spørgsmål?

```
my_append([],List,List).  
my_append([H|Tail], X, [H|Newtail]) :-  
    my_append(Tail, X, Newtail).
```

```
my_append([a,b,c], Y, [a,b,c,d,e]).  
Y = [d,e]  
Yes
```

- Spørgsmålet matcher ikke den første regel, men derimod den anden.
- Det initiale mål erstattes af den anden regel's krop med
 - $H = a$, $Tail = [b,c]$, $X = Y$, $Newtail = [b,c,d,e]$.
- Bevis: `my_append([b,c], Y, [b,c,d,e])`.
- Ovenstående mål erstattes af den anden regel's krop med
 - $H' = b$, $Tail' = [c]$, $X' = Y$, $Newtail' = [c,d,e]$.
- Bevis: `my_append([c], Y, [c,d,e])`.
- Ovenstående mål erstattes af den anden regel's krop med
 - $H'' = c$, $Tail'' = []$, $X'' = Y$, $Newtail'' = [d,e]$.
- Bevis: `my_append([], Y, [d,e])`.
- Første regel matcher: $Y = [d,e]$.
- Svar: $Y = [d,e]$.

PS3 - Logikprogrammering

Noter

Lige som tidligere ser vi på de konkrete trin i bevisførelsen for, at der findes en liste Y , som appended til $[a, b, c]$ giver $[a, b, c, d, e]$.

Bemærk at vi som følge af rekursionen i anvendelsen af regler introducerer mærkede og dobbeltmærkede variable.

Konventionelle beregninger i Prolog.

- Prolog indeholder en række *operatorer*, som kan indgå i konventionelle *udtryk*.
 - Boolske operatorer: Passer godt til paradigmet.
 - Aritmetiske operatorer:
 - Var *is* aritmetisk udtryk.
 - Variable bindes til værdin af det aritmetiske udtryk.

```
fak(0,1).  
fak(N,R) :- M is N-1, fak(M,S), R is N * S.
```

```
?- fak(5,Res).  
Res = 120  
Yes
```

Noter

Vurdering af paradigmet.

- Den rene logikprogrammering retter sig primært mod problemer, der kan løses ved søgning i en videnbase.
 - Prolog tillader andre udtryksformer (såsom beregning af ikke-logiske udtryk) - på bekostning af paradigmets renhed.
- Hvis man ønsker at påvirke søgestrategien - og dermed logikprogrammets effektivitet - må man bekymre sig om de implementationsdetaljer i søgningen.
 - Den rene, deklarative tankegang fortrænges.
- Logikprogrammering orienterer sig mod det *rige interaktionsparadigme*.

Noter