Logikprogrammering

•Kännetecken och förutsättningar
•Prolog
•Förtjänster
•Begränsningar
•Praktiska tillämpningar

Kännetecken

- Hög abstraktionsnivå
- Deklarativt, ej proceduralt
- Specificerar önskade resultat snarare än hur de ska produceras
- Bygger på formell logik, relationer

Relationer

- Många-till-många relation
- Jfr avbildningar, många-till-en
  - Alltså ett värde x i S till ett y i T
- Men relationer, x till flera y i T

Exempel på relation

- S=[1,2,3], T=[4,5,6], rel="<"
- 1<4, 1<5, 1<6
- 2<4, 2<5, 2<6
- 3<4, 3<5, 3<6

Förfrågningar

- Är R(a,b) sann?
- För vilka y är R(a,y) sann?
- För vilka x är R(x,b) sann?
- För vilka x,y är R(x,y) sann?

Varianter

- Unary (unär) R(x)
- Binary (binär) R(x,y)
- Ternary (ternär) R(x,y,z)
- Flervärda

Exempel

- unär: mamma(lena)
- binär: förälder(lena,victor)
- ternär (trevärd):
  tregenerationer(kjell,lena,victor)

Logik och logikprogrammering

- Många relationer som enkelt "ryms" inom logiken kan inte implementeras, logikprogrammering klarar *inte* att uttrycka den matematiska logiken i dess fulla kraft.
- Horn-klausuler är lösningen

Hornklausuler (forts.)

A_0 *if* A_1 *and ... and* A_n

- Varje A_i har formen $R_i(...)$ där R_i är namnet på en relation
- Betyder att om $A_1...A_n$ är sann så är A_0 sann
- Om någon A_i är falsk innebär det ej att A_0 är falsk
- Specialfall när n = 0
  A_0
  - A_0 är ovillkorligen sann

Logikprogram

- Ett logikprogram består av en samling Hornklausuler
- Beräkningen av ett program består i att testa ett givet antagande/mål (*A*)
  - Om vi kan härleda ur programmet att *A* är sann, säger vi att *A* lyckas (succeeds), annars misslyckas vi (fail)
  - Om vi misslyckas innebär det inte att *A* är falsk
- Hur härledningen går till exakt är ej definierat i generell logikprogrammering

Resolution

- Resolution är ett sätt att härleda svar ur ett logikprogram (resolution=upplösning; sönderdelning)
- Hornklausuler och resolution
  - Gör logikprogram implementerbara
  - Tillåter implementationen att bli tämligen effektiv

Resolution

- Härled *A*
- Det existerar en klausul A_0 som matchar *A* => *A* är sann
- Det existerar en klausul A_0 *if* A_1 *and ... and* A_n sådan att A_0 matchar *A* =>
  - fortsätter härleda A_1 till A_n som separata delmål
  - Om alla lyckas drar vi slutsatsen att *A* är sann
  - Om något A_i misslyckas måste vi backa (backtrack), d v s ge upp den klausulen och testa nästa
- Om inga klausuler matchar *A* har vi misslyckats

Exempel

Man gör en förfrågan:

kvinna(lena)

I databasen hittar vi:

mamma(lena)

kvinna(x) (mamma(x)

Backtracking

- Relationer i logikspråk har samma betydelse som funktioner och procedurer i andra språk
- Men det är backtracking som ger logikspråk deras stora uttrycksfullhet och kraftfullhet
  - Men det kostar i tid och minneskrav

Exempel

Tänk följande mål:

`male(X) ∩ parent(X, lotta)`

Man hittar: `male(mikael)`

och går vidare för att hitta

`parent(mikael, lotta)`

Om detta inte lyckas blir backtracking aktuellt, och ny matchning med

Prolog

- Prolog utvecklades gradvis under 70-talet och ett flertal dialekter uppstod
  - Numera är Edinburgh-dialekten accepterad som standard
- Ett Prologprogram består av en samling Hornklausuler som definierar relationer

Värden

- Prologs värden är tal, atomer och strukturer
- Atomer
  - Atomer har inga andra egenskaper än att de går att skilja från varandra. Ex `röd`, `lisa`, `kroton`....
- Strukturer
  - Strukturer är taggade tuppler, t ex `data(2000,jan,1)`
  - Taggen används för att skilja strukturer från varandra, som representerar olika data men har samma värden, t ex `point(2,3)`, `rational(2,3)`.

Listor och strängar

- Listor
  - En delmängd till strukturer där [] representerar tomma listan och '.' är strukturtaggen:

```
[1,2,3,4] = .(1,.(2,.(3,.(4,[]))))
[x|xs] = .(x,xs)
```

- Strängar
  - Listor av heltal
  - Varje bokstav representeras av heltal,

```
"Olof" =  [79, 108, 111, 102]
```

Typer

- Prolog är otypat
- Tal, atomer och strukturer kan användas om varandra
- De kan jämföras med varandra m h a "="
  - ger "false" om de är inte är lika
- Endast aritmetiska operationer är typade

Termer och variabler

- Term
  - En term är en variabel, en numerisk literal, en atom eller strukturaggregat
  - Termer är argument till relationerna $R(T_1,\ldots,T_n)$ där $T_1 \ldots T_n$ är termer
- Variabler
  - Prologs variabler betecknar ett fixt men okänt värde
  - Börjar alltid på stor bokstav för att skiljas från strukturer och atomer.
  - En variabel deklareras implicit genom att dess förekomst i klausuler och dess räckvidd är just

Klausuler och relationer

- Klausuler

  A_0 :- A_1, A_2, ..., A_n.
  - ":-" står för *if* och "," står för *and*
  - A_0 matchar ett påstående A om det existerar en substitution av termer för variabler sådan att A_0 och A blir lika
  - *age(P,Y)* matchar *age("David",42)* under
    {P = "David",Y = 42}
    age(P,Y) matchar *age("Olof",A)* under
    {P = "Olof", Y = A}
- Räckvidden för varje relation är hela programmet

Exempel

```
star(sun).
star(sirius).
star(betelgeuse).

?- star(sun).
Yes
?- star(jupiter).
False
orbits(mercury,sun).
orbits(venus,sun).
```

Exempel (forts.)

```
.....
?- orbits(venus,sun).
Yes
?- orbits(B,sun).
B = mercury
B = venus
```

Exempel (forts.)

```
....
planet(B) :- orbit(B,sun).
?- planet(venus).
Yes
?- planet(B).
B = mercury
B = venus
```

Exempel (forts.)

```
.....
satellite(B) :-
orbits(B,P),planet(P).
solar(sun).
solar(P) :- planet(P).
solar(B) :- satellite(B).
solar(sun).
solar(moon).
```

Exempel (forts.)

```
.....
solar(B). B = sun, mercury,
venus,...,moon...
```

- Fråga: **existerar** det instanser så att frågan blir sann
- Klausuler: A_0 är sant för **alla** variabler $A_1 ... A_n$ som är sanna

Rekursiva klausuler:

```
member(X,[X|Ys]).
member(X,[Y|Ys]) :- member(X,Ys).
```

"Sluten värld"-antagande

- Prolog utgår från att alla relevanta fakta finns i programmet
- Ett påstående misslyckas om vi inte kan härleda att det är sant
- Det innebär inte att det är falskt utan kan också innebära att det är okänt
- Detta innebär att Prolog inte skiljer på falsk och okänt
- Det medför att negationen av okänt blir

Kontroll

- I princip ska den ordning i vilken resolutionen görs inte spela någon roll för resultatet, endast ordningen på resultaten
- I Prolog är ordningen för resolutionen av avgörande betydelse dels för att överhuvudtaget få resultat men även för effektiviteten

Exempel

```
neighbor(A,B) :- neighbor(B,A).
neighbor(A,B) :- planet(A),orbits(B,A).
```

- Om det icke rekursiva fallet används först hela tiden uppstår inga problem
- Om det rekursiva fallet används först uppstår oändlig rekursion

Ordningen för resolutionen

- Beteendet hos Prologprogram beror på ordningen för resolutionen
- I Prolog testas högerledet från vänster till höger i klausulerna och om det finns flera klausuler för en relation testas de i ordning från första till sista
- Det gör att Prologprogram är deterministiska, även ordningen för resultaten är bestämd
- Ett sätt för programmeraren att styra ordningen är med *cut (!)* som skär av möjligheten till backtracking
- Det kan ge effektivare program, men också förändra betydelsen, används med förstånd

Förtjänster

- Hög abstraktionsnivå
- Logik - en bra grund för välstrukturerade program, leder till färre fel i programmen och därmed mindre underhållsbehov
- Ger stöd för parallellitet
- Kompakta program

Begränsningar

- Långsamt
- Risk för oändliga loopar
- Man kan inte vara säker på att något är falskt
- Negationsproblemet

Praktiska tillämpningar

- RDBMS
  (Relational Database Management Systems)
- Expertsystem
- Naturligt språk
- Utbildning