

Logično programiranje

Logična pravila

Do sedaj smo spoznali *ukazno* in *deklarativno* programiranje:

- Pri ukaznem programiranju na izvajanje programa gledamo kot na zaporedje akcij, ki spreminjajo stanje sistema (vrednosti spremenljivk).
- Pri deklarativnem programiranju je program izraz, ki se preračuna v končno *vrednost*.

Logično programiranje izhaja iz ideje, da je izvajanje programa **iskanje dokaza**. Da bomo to razumeli, najprej ponovimo nekaj osnov logike.

Hornove formule

V logiki prvega reda lahko zapišemo formule sestavljene iz konstant $\perp$, $\top$, veznikov $\wedge$, $\vee$, $\Rightarrow$, $\neg$ in kvantifikatorjev $\forall$ (za vsak), $\exists$ (obstaja). Take formule so lahko precej zapletene in niso primerne za logično programiranje, zato se omejimo na tako imenovane **Hornove formule**, ki so oblike

$$\forall x_1, \dots, x_i . (\phi_1 \wedge \phi_2 \wedge \dots \wedge \phi_j \Rightarrow \Psi)$$

Tu so $\phi_1, \dots, \phi_j$ in Ψ **osnovne formule**, se pravi vsaka od njih je oblike

$$p(t_1, \dots, t_i)$$

kjer je p **relacijski simbol** in $t_1, \dots, t_i$ **termi**. Nadalje je term izraz, ki ga lahko sestavimo iz konstant, funkcijskih simbolov in spremenljivk.

Poseben primer Hornove formule je **dejstvo**, ki ga dobimo pri $j = 0$:

$$\forall x_1, \dots, x_i . \Psi$$

Drugi primer je formula brez kvantifikatorjev (v kateri ni spremenljivk, samo konstante), ki ga dobimo pri $i = 0$:

$$\phi_1 \wedge \phi_2 \wedge \dots \wedge \phi_j \Rightarrow \Psi$$

Poglejmo si nekaj primerov.

Primer

Hornova formula

$$\forall a . (\text{pes}(a) \Rightarrow \text{zival}(a))$$

pove, da so psi živali: "za vsak a , če je a pes, potem je a žival".

Primer

Hornova formula

$\forall x y z . (\text{otrok}(x, y) \wedge \text{otrok}(y, z) \wedge \text{zenska}(z) \Rightarrow \text{babica}(x, z))$

pravi: "za vse (osebe) x, y, z , če je x otrok od y in y otrok od z in je z ženska, potem je z babica od x ".

Vzgojen primer

Formula

$\forall x y z . \text{otrok}(x, z) \wedge \text{otrok}(y, z) \wedge \text{zenska}(x) \wedge \text{zenska}(y) \Rightarrow \text{sestra}(x, y)$

ne pomeni " x in y sta sestri" ampak " x in y sta sestri ali polsestri, ali pa sta enaka".

Primer

S Hornovimi formulami lahko izrazimo tudi matematična dejstva. Peanova aksioma za seštevanje se glasita

$\forall n . n + 0 = n$
 $\forall k m . k + \text{succ}(m) = \text{succ}(k + m)$

Pravzaprav v Prologu ni funkcij, težko definiramo vsoto kot funkcijo! Definirati moramo *relacijo*, ki predstavlja funkcijo:

Pravimo, da relacija R predstavlja funkcijo f , če je $f(x) = y \Leftrightarrow R(x, y)$.

Za funkcijo seštevanja: namesto operacije $+$ bomo definirali relacijo *vsota*, da bo veljalo:

$x + y = z \Leftrightarrow \text{vsota}(x, y, z)$

S Hornovima formulama ju zapišemo takole, pri čemer $\text{vsota}(x, y, z)$ beremo "vsota x in y je z ":

$\forall n . \text{vsota}(n, \text{zero}, n)$
 $\forall k m n . \text{vsota}(k, m, n) \Rightarrow \text{vsota}(k, \text{succ}(m), \text{succ}(n))$

Prva formula očitno ustreza prvemu aksiomu, druga pa je ekvivalentna

$\forall m n k . k + m = n \Rightarrow k + \text{succ}(m) = \text{succ}(n)$

kar je ekvivalentno drugemu aksiomu (premisli zakaj!).

Naloga

S Hornovimi formulami zapiši Peanova aksioma za množenje:

$\forall n . n \cdot 0 = 0$
 $\forall k m . k \cdot \text{succ}(m) = k + k \cdot m$

Uporabi relacijo *vsota* iz prejšnjega primera, ter relacijo *zmnozek* (x, y, z) , ki ga beremo "zmnožek x in y je z ".

Formule, ki niso Hornove

Nekaterih dejstev s Hornovimi formulami ne moremo izraziti, na primer negacije ne eksistenčnih formul $\exists x . \dots$.

Sistematično iskanje dokaza

Denimo, da imamo Hornove formule in želimo vedeti, ali iz njih sledi dana izjava. Kako bi *sistematično* poiskali dokaz?

Primer

Najprej pogledjmo primer brez kvantifikatorjev. Ali iz Hornovih formul

1. $x \wedge y \Rightarrow c$
2. $A \wedge B \Rightarrow C$
3. $x \Rightarrow B$
4. $A \Rightarrow B$
5. A

sledi c ?

Iskanja dokaza se lotimo sistematično. Katera od formul bi lahko pripeljala do dokaza izjave c ? Prva ali druga. Poskusimo obe:

- če uporabimo $x \wedge y \Rightarrow c$, c prevedemo na *podnalogi* x in y . A tu se zatakne, ker o x in y ne vemo nič pametnega.
- če uporabimo $A \wedge B \Rightarrow C$, c prevedemo na *podnalogi* A in B :
 - dokažimo A : to velja zaradi 5. formule
 - dokažimo B : uporabimo lahko 3. ali 4. formulo. Tretja ne deluje, četrta pa dokazovanje prevede na podnalogo A , ki velja.

Primer

Ali iz

1. $\forall x y . \text{otrok}(x, y) \Rightarrow \text{mlajsi}(x, y)$
2. $\text{otrok}(\text{miha}, \text{mojca})$

sledi $\text{mlajsi}(\text{miha}, \text{mojca})$? Če v prvi formuli vzamemo $x = \text{miha}$ in $y = \text{mojca}$, lahko nalogo prevedemo na $\text{otrok}(\text{miha}, \text{mojca})$. To pa velja zaradi druge formule.

Primer

Ali iz

1. $\forall x . \text{sodo}(x) \Rightarrow \text{liho}(\text{succ}(x))$
2. $\forall y . \text{liho}(y) \Rightarrow \text{sodo}(\text{succ}(y))$
3. $\text{sodo}(\text{zero})$

sledi $\text{sodo}(\text{succ}(\text{succ}(\text{zero})))$? Tokrat zapišimo bolj sistematično postopek iskanja:

- dokaži $\text{sodo}(\text{succ}(\text{succ}(\text{zero})))$
- uporabimo drugo formulo, za y vstavimo $\text{succ}(\text{zero})$ in dobimo nalogo
- dokaži $\text{liho}(\text{succ}(\text{zero}))$
- uporabimo prvo formulo, za x vstavimo zero in dobimo nalogo

- dokaži `sodo(zero)`
- to velja zaradi tretje formule.

V splošnem bomo morali rešiti nalogo **združevanja**: poišči take vrednosti spremenljivk, da sta dani formuli enaki*. Podobno nalogo smo že reševali, ko smo obravnavali parametrični polimorfizem, kjer smo izenačevali tipe.

Logično programiranje

V logičnem programiranju je program podan s

- seznamom **pravil** $H_1, \dots, H_i$ in
- **poizvedbo** G

Pravila so Hornove formule. Poizvedba je formula oblike

$$\exists y_1, \dots, y_j . p(y_1, \dots, y_j)$$

Zanima nas, ali poizvedba sledi iz pravil. Poizvedbo G predelamo na poizvedbe in **iščemo v globino**, takole:

1. V seznamu $H_1, \dots, H_i$ poišči prvo formulo, ki je oblike

$$\forall x_1, \dots, x_i . \phi_1(x_1, \dots, x_i) \wedge \dots \wedge \phi_r(x_1, \dots, x_i) \Rightarrow \Psi(x_1, \dots, x_i),$$

katere sklep Ψ je **združljiv** z G . To pomeni da lahko za $y_1, \dots, y_j$ vstavimo neke vrednosti $u_1, \dots, u_j$ in za $x_1, \dots, x_i$ neke vrednosti $v_1, \dots, v_i$, da sta formuli

$$p(u_1, \dots, u_j)$$

in

$$\Psi(v_1, \dots, v_i)$$

enaki.

Opomba: možno je, da izbira vrednosti $u_1, \dots, u_j$ in $v_1, \dots, v_i$ ni enolična. V tem primeru izberemo *najbolj splošne vrednosti*. To naredimo s postopkom združevanja (angl. unification), ki smo ga spoznali pri parametričnem polimorfizmu.

2. Poizvedbo smo predelali na poizvedbe $\phi_1(v_1, \dots, v_i), \dots, \phi_r(v_1, \dots, v_i)$, ki jih rešujemo po vrsti rekurzivno. (Če se v teh poizvedbah pojavljajo spremenljivke, jih obravnavamo, kot da smo jih kvantificirali z $\exists$.)

Primer

Poglejmo si še enkrat primer, ko imamo Hornove formule

1. `sodo(zero)`
2. $\forall x . \text{sodo}(x) \Rightarrow \text{liho}(\text{succ}(x))$
3. $\forall y . \text{liho}(y) \Rightarrow \text{sodo}(\text{succ}(y))$

in poizvedbo

$$\exists z . \text{liho}(z)$$

Ali lahko združimo $\text{sodo}(\text{zero})$ in $\text{liho}(z)$? Ne.

Ali lahko združimo $\text{liho}(\text{succ}(x))$ in $\text{liho}(z)$? Poskusimo s postopkom združevanja: Enačbo

$$\text{liho}(\text{succ}(x)) = \text{liho}(z)$$

prevedemo na enačbo

$$\text{succ}(x) = z$$

Dobili smo rešitev za z in ni več enačb, torej je x poljuben. Torej uporabimo drugo pravilo, ki prevede nalogo na

$$\exists x . \text{sodo}(x)$$

Ali lahko to združimo s prvo formulo? Poskusimo rešiti

$$\text{sodo}(\text{zero}) = \text{sodo}(x)$$

Rešitev je $x = \text{zero}$. Ker je prva formula dejstvo, ni nove podnaloge.

Rešitev se glasi: $x = \text{zero}$, $z = \text{succ}(x)$. Končna rešitev je torej

$$z = \text{succ}(\text{zero})$$

Dokazali smo, da res obstaja liho število, namreč $\text{succ}(\text{zero})$.

Naloga

Če v prejšnjem primeru zamenjamo vrstni red pravil,

1. $\forall x . \text{sodo}(x) \Rightarrow \text{liho}(\text{succ}(x))$
2. $\forall y . \text{liho}(y) \Rightarrow \text{sodo}(\text{succ}(y))$
3. $\text{sodo}(\text{zero})$

potem poizvedba

$$\exists z . \text{liho}(z)$$

privede do neskončne zanke (ker iščemo v globino in vedno uporabimo prvo pravilo, ki deluje). Namreč, z uporabo prvega pravila dobimo poizvedbo

$$\exists x . \text{sodo}(x)$$

nato z uporabo drugega pravila ($x = \text{succ}(y)$)

$$\exists y . \text{liho}(y)$$

nato z uporabo prvega pravila

$$\exists u . \text{sodo}(u)$$

in tako naprej. Tretje pravilo nikoli ne pride na vrsto!

Prolog

Prolog je programski jezik, v katerem logično programiramo. Ima nekoliko nenavadno sintakso:

- namesto $A \wedge B$ pišemo A, B
- namesto $A \vee B$ pišemo $A; B$
- namesto $A \Rightarrow B$ pišemo $B :- A$ (pozor, zamenjal se je vrstni red, $B \Leftarrow A!$)
- kvantifikatorjev $\forall x \dots$ in $\exists x \dots$ ne pišemo, ampak **kvantificirane spremenljivke pisemo z velikimi črkami**
- **konstante, predikate in funkcije pišemo z malimi črkami.**

Na koncu vsake formule zapišemo piko.

Predelajmo primer iz prejšnjega razdelka v Prolog. Najprej v datoteko [even_odd.pl](#) spravimo pravila (pri čemer pravila za sodo zložimo skupaj, da se ne pritožuje):

```
sodo(zero).
sodo(succ(Y)) :- liho(Y).
liho(succ(X)) :- sodo(X).
```

Datoteko naložimo v interaktivno zanko. Ta nam omogoča, da vpišemo poizvedbo in dobimo odgovor:

```
?- liho(Z).
Z = succ(zero) ;
Z = succ(succ(succ(zero))) ;
Z = succ(succ(succ(succ(succ(zero))))) .
```

Ko nam prolog poda odgovor, lahko z znakom `;` zahtevamo, da išče še naprej. Z znakom `.` zaključimo iskanje.

Naloga

Ali se prolog res spusti v neskončno zanko, če zamenjamo vrsti red pravil za sodo?

Naloga

Na svoj računalnik si namesti [SWI Prolog](#) in poženi zgornji program.

Seznami

Predstavitev seznamov v Prologu

Kako bi v Prologu naredili sezname? V Ocamlu smo jih definirali kot induktivni tip:

```
type 'a list = Nil | Cons of 'a * 'a list
```

Na primer, `Cons(a, Cons(b, Cons(c, Nil)))` je seznam z elementi `a`, `b` in `c`.

V Prologu ni tipov, lahko pa uporabljamo poljubne konstante in konstruktorje, le z malimi črkami jih je treba pisati. Torej lahko sezname še vedno predstavljamo z `nil` in `cons`.

Seznamov ni treba vnaprej definirati, se pravi, ni treba razlagati, kaj sta `nil` in `cons`. Prolog ju obravnava kot simbola, s katerimi lahko tvorimo izraze. Seznam `[a; b; c]` zapišemo `cons (a, cons (b, cons (c, nil)))`.

Opomba: Prolog ima tudi vgrajene sezname, glej spodaj.

Relacija `elem`

Da bomo dobili občutek za moč logičnega programiranja, definirajmo nekaj funkcij za delo s seznamami.

Naš prvi program je relacija, ki ugotovi, ali je dani x pripada danemu seznamu L :

```
elem(X, cons(X, _)).  
elem(X, cons(_, L)) :- elem(X, L).
```

Poiskujemo (glej datoteko [list.pl](#)):

```
?- elem(a, cons(b, cons(a, cons(c, cons(d, cons(a, nil)))))).  
true ;  
true ;  
false.
```

Zakaj smo dvakrat dobili `true` in nato `false`?

Vprašamo lahko tudi, kateri so elementi danega seznama:

```
?- elem(X, cons(a, cons(b, cons(a, cons(c, nil)))).  
X = a ;  
X = b ;  
X = a ;  
X = c ;  
false.
```

In celo, kateri seznam vsebuje dani element!

```
?- elem(a, L).  
L = cons(a, _3234) ;  
L = cons(_3232, cons(a, _3240)) ;  
L = cons(_3232, cons(_3238, cons(a, _3246))) ;  
L = cons(_3232, cons(_3238, cons(_3244, cons(a, _3252)))) ;  
L = cons(_3232, cons(_3238, cons(_3244, cons(_3250, cons(a, _3258))))) ;  
L = cons(_3232, cons(_3238, cons(_3244, cons(_3250, cons(_3256, cons(a, _3264))))) .
```

Prolog je ustvaril pomožne spremenljivke `_xyzw`, s katerimi označi poljubne terme.

Relacija `join`

Funkcijo, ki stakne seznama predstavimo s trimestno relacijo `join`:

`join(X, Y, Z)` pomeni, da je Z enak stiku seznamov X in Y .

Zapišimo pravila zanjo:

```
join(nil, Y, Y).  
join(cons(A, X), Y, cons(A, Z)) :- join(X, Y, Z).
```

To je podobno funkciji, ki bi jo definirali v OCamlu:

```
let rec join x y=  
  match (x, y) with  
  | (Nil, y) -> y  
  | (Cons (a, x), y) ->  
    let z = join x y in  
    Cons (a, z)
```

Takole izračunamo stik seznamov `cons(a, cons(b, nil))` in `cons(x, cons(y, cons(z, nil)))`:

```
?- join(cons(a, cons(b, nil)), cons(x, cons(y, cons(z, nil))), Z).
```

```
Z = cons(a, cons(b, cons(x, cons(y, cons(z, nil))))).
```

Vgrajeni sezname

Prolog že ima vgrajene sezname:

- $[e_1, e_2, \dots, e_i]$ je seznam elementov $e_1, e_2, \dots, e_i$.
- $[e \mid \ell]$ je seznam z glavo e in repom ℓ
- $[e_1, e_2, \dots, e_i \mid \ell]$ je seznam, ki se začne z elementi $e_1, e_2, \dots, e_i$ in ima rep ℓ .

Za delo s sezname je na voljo [knjižnica lists](#), ki jo naložimo z ukazom

```
:- use_module(library(lists)).
```

Ta že vsebuje relaciji `member` (ki smo jo zgoraj imenovali `elem`) in `append` (ki smo jo zgoraj imenovali `join`). Preizkusimo:

```
?- append([a,b,c], [d,e,f], Z).  
Z = [a, b, c, d, e, f].
```

Lahko pa tudi vprašamo, kako razbiti seznam $[a, b, c, d, e, f]$ na dva podeznama:

```
?- append(X, Y, [a,b,c,d,e,f]).  
X = [],  
Y = [a, b, c, d, e, f] ;  
X = [a],  
Y = [b, c, d, e, f] ;  
X = [a, b],  
Y = [c, d, e, f] ;  
X = [a, b, c],  
Y = [d, e, f] ;  
X = [a, b, c, d],  
Y = [e, f] ;  
X = [a, b, c, d, e],  
Y = [f] ;  
X = [a, b, c, d, e, f],  
Y = [] ;  
false.
```

Enakost in neenakost

Včasih v Prologu potrebujemo enakost in neenakost. Enakost pišemo $s = t$ in neenakost $s \neq t$.

Zanimiv primer

Če bo čas, si bomo ogledali, kako v Prologu implementiramo tolmač za preprost ukazni programski jezik, glej [comm.pl](#).