Še nekaj o signaturah

Zadnjič smo se spraševali, kako narediti signaturo za grupo v Javi. V SML se glasi takole:

```
signature GROUP =
sig
  type t
  val e : t
  val mul : t -> t -> t
  val inv : t -> t
end
```

Aljaž Eržen mi je poslal idejo, ki dobro deluje. V Javi moramo signaturo *parametrizirati* s tipom, namesto da ga vstavimo v singaturo:

```
public interface Group<T> {
    T e();
    T mul(T a, T b);
    T inv(T a);
}
```

Nato lahko definiramo grupo Z_3 takole:

```
public class Z3 implements Group<Z3> {
    ⋮
}
```

To malce spominja na operacijo curry v funkcijskem programiranju:

```
T × ⋯ : Structure
```

```
Type → ... : Structure
```

Curry:

```
A × B → C
```

```
A → (B → C)
```

And now for something completely different.

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 med izvajanjem predelamo 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 $\bot$, $\top$, veznikov $\land$, $\lor$, $\Rightarrow$, $\neg$ in kvantifikatorjev $\forall$, $\exists$. Take formule so lahko precej zapletene in niso primerne za logično programiranje, kjer se omejimo na tako imenovane **Hornove formule**, ki so oblike

$$\forall x_1, \dots, x_i . (\Phi_1 \land \Phi_2 \land \cdots \land \Phi_j \Rightarrow \Psi)$$

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**:

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

Drugi primer je formula brez kvantifikatorjev (v kateri ni spremenljivk, samo konstante):

$$\Phi_1 \land \Phi_2 \land \cdots \land \Phi_j \Rightarrow \Psi$$

To je res Hornova formula, če si predstavljamo, da je oblike

$$\forall x_1, \dots, x_i . (\Phi_1 \land \Phi_2 \land \cdots \land \Phi_j \Rightarrow \Psi)$$

kjer je `j = 0`.

Poglejmo si nekaj primerov.

Primer

Hornova formula

```
∀ a . (pes(a) ⇒ zival(a))
```

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

Primer

Hornova formula

```
∀ x y z . (otrok(x, y) ∧ otrok(y, z) ∧ zenska(z) ⇒ `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

```
∀ x y z . otrok(x, z) ∧ otrok(y, z) ∧ zenska(x) ∧ zenska(y) ⇒ sestra(x, y)
```

pomeni *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

```
∀ n . n + 0 = n
∀ k m . k + succ(m) = 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 ⇔ R (x, y)`.

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

```
x + y = z ⇔ vsota(x, y, z)
```

S Hornovima formulama ju zapišemo takole, pri čemer `vsota(x,y,z)` beremo "vsota `x` in `y` je `z`":

```
∀ n . vsota(n, zero, n)
∀ k m n . vsota(k, m, n) ⇒ vsota(k, succ(m), succ(n))
```

Prva formula očitno ustreza prvemu aksiomu, druga pa je ekvivalentna

```
∀ m n k . k + m = n ⇒ k + succ(m) = succ(n)
```

kar je ekvivalentno drugemu aksiomu (premisli zakaj!).

Naloga

S Hornovimi formulami zapiši Peanova aksioma za množenje:

```
∀ n . n · 0 = 0
∀ k m . k · succ(m) = k + k · 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 `∃ x . ⋯`.

Sistematično iskanje dokaza

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

Primer

Najprej poglejmo primer brez kvantifikatorjev. Ali iz Hornovih formul

1. `X ∧ Y ⇒ C`
2. `A ∧ B ⇒ C`
3. `X ⇒ B`
4. `A ⇒ 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 ∧ Y ⇒ C`, `C` prevedemo na *podnalogi* `X` in `Y`. A tu se zatakne, ker o `X` in `Y` ne vemo nič pametnega.

- če uporabimo `A ∧ B ⇒ 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. `∀ x y . otrok(x, y) ⇒ mlajsi(x, y)`
2. `otrok(miha, mojca)`

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

Primer

Ali iz

1. `∀ x . sodo(x) ⇒ liho(succ(x))`
2. `∀ y . liho(y) ⇒ sodo(succ(y))`

3. `sodo(zero)`

sledi `sodo(succ(succ(zero)))`? Tokrat zapišimo bolj sistematično postopek iskanja:

- dokaži `sodo(succ(succ(zero)))`
- uporabimo drugo formulo, za `y` vstavimo `succ(zero)` in dobimo nalogo
- dokaži `liho(succ(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** $H_1, \ldots, H_i$ in
- **poizvedbo** G

Pravila so Hornove formule. Poizvedba je formula oblike

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

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

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

   $$\forall\, x_1, \ldots, x_i \,.\, \Phi_1(x_1, \ldots, x_i) \land \cdots \land \varphi_r(x_1, \ldots, x_i) \Rightarrow \Psi(x_1, \ldots, x_i),$$

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

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

   in

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

   enaki.

   Opomba: možno je, da izbira vrednosti $u_1, \ldots, u_j$ in $v_1, \ldots, v_i$ 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 $\Phi_1(v_1, \ldots, v_i), \ldots, \Phi_r(v_1, \ldots, v_i)$, ki jih rešujemo po vrsti rekurzivno. (Če se v teh poizvedbah

pojavljajo spremenljivke, jih obravnavamo, kot da smo jih kvantificirali z ∃.)

Primer

Poglejmo si še enkrat primer, ko imamo Hornove for

1. `sodo(zero)`
2. `∀ x . sodo(x) ⇒ liho(succ(x))`
3. `∀ y . liho(y) ⇒ sodo(succ(y))`

in poizvedbo

`∃ z . liho(z)`

Ali lahko združimo `sodo(zero)` in `liho(z)`? Ne.

Ali lahko združimo `liho(succ(x))` in `liho(z)`? Poskusimo s postopkom združevanja: Enačbo

`liho(succ(x)) = liho(z)`

prevedemo na enačbo

`succ(x) = z`

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

`∃ x . sodo(x)`

Ali lahko to združimo s prvo formulo? Poskusimo rešiti

`sodo(zero) = sodo(x)`

Rešitev je `x = zero`. Ker je prva formula dejstvo, ni nove podnaloge.

Rešitev se glasi: `x = zero`, `z = succ(x)`. Končna rešitev je torej

`z = succ(zero)`

Dokazali smo, da res obstaja liho število, namreč `succ(zero)`.

Naloga

Če v prejšnjem primeru zamenjamo vrstni red pravil,

1. `∀ x . sodo(x) ⇒ liho(succ(x))`
2. `∀ y . liho(y) ⇒ sodo(succ(y))`
3. `sodo(zero)`

potem poizvedba

`∃ z . liho(z)`

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

```
∃ x . sodo(x)
```

nato z uporabo drugega pravila

```
∃ y . liho(y)
```

nato z uporabo prvega pravila

```
∃ u . 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 ∧ B` pišemo `A, B`
- namesto `A ∨ B` pišemo `A; B`
- namesto `A ⇒ B` pišemo `B :- A` (pozor, zamenjal se je vrstni red, `B ⇐ A`!)
- kvantifikatorjev `∀ x …` in `∃ x …` pišemo **spremenljivke 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 datoeko 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 interkativno 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 oddgovor, 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 SML smo spoznali, da jih lahko definiramo sami:

```
datatype '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`.

Relacija `elem`

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

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).
```

Poiskusimo:

```
?- 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 seznami vsebujejo 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)))))) .
```

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 SML:

```
fun join nil y = y
  | join (cons(a, x)) y =
    let val z = join x y
    in cons (a, z)
    end
```

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 seznami

Prolog že ima vgrajene sezname:

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

Za delo s seznami je na voljo [knjižnica <code>lists</code>], 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 \= t`.

Zanimiv primer

Če bo čas, si bomo ogledali, kako v Prologu implementiramo tolmač za preprost ukazni programski jezik.

Programiranje z omejitvami

To se bomo učili naslednjič.