Típusos Prolog

Típus fogalma

Egy adott paraméter pozíción értelmes tömör Prolog kifejezések halmaza

Prologban általában nincs típushiba, a hiba hatására a hívás meghiúsul.

```
tarcsaz(bejegyzes(Nev,Szam),Szam).

tarcsaz(b('Bela',2131233),Tone).
no
```

Hibajelzést csak beépített típusok esetében kapunk

```
X is 1+alma
Domain error in argument 2 of is/2
Expected expression, found alma
Goal: _79 is 1+alma
```

Sicstus Prologban nincs típusdeklaráció (csak az olvasást megkönnyítendő fejkomment).

Mercury típusnyelve

Elsősorban adatstruktúrák kialakítására alkalmas.

Beépített típusok

Ismert primitív típusok

```
char int float string number
```

Predikátum típusok:

A predikátumok argumentumainak típusát határozza meg

```
pred, pred(T)

:- pred is_all_uppercase(string)
```

Függvény típusok:

A függvények argumentumainak és visszatérési értékének típusát határozza meg

```
func(T1,T2)=T

:- func length(list(T))=int
```

Tuple – „n-esek”

`:- type {Arg1, Arg2, …, ArgN} ---> {{Arg1, Arg2, …, ArgN}}`

Miért szükséges az két egymásba ágyazott kapcsos zárójel ???

`{}` Általános funktornév prefix alakban:

`'{}'(Arg1, .. ArgN)`

De mint „sznitaktikai édesítőszer”:

`{Arg1, … ArgN} ≡ '{}'((Arg1, … , ArgN))`

Tehát egyargumentumú funktornak képződik le

A típus deklarációban külső kapcsos záró jelek mentesítik a belsőket az alól, hogy a prolog édesítőszernek értelmezze azokat.

Univerzális típus

Standard library **`std_util`** moduljában
`type_to_univ/2` és **`univ_to_type_2`** predikátumokkal konvertálhatóak.

Heterogén kollekciókban.

Felhasználói típusok

Új típusokat a **`:-type`** deklarációval hozunk létre

Megkülönböztetett unió

„becsomagolt” adatok

Megkülönböztett unió: véges sok különböző funktorú típusból képzett halmaz uniója

Mercury-ban csak megkülönböztetett uniók használhatóak. Sicstus Prologban megengedett legfeljebb kevésbé hatékony

```
:-type t--->s1(t1,t2,...,tn1);...; sm(t1,t2,...,tnm).
```

`ni`=0 esetén constans

```
:-type tree ---> leaf(integer);node(tree,tree).

 sum_tree(Tree, S):-
     Tree=leaf(Value),
     S=Value.

 sum_tree(Tree, S):-
     Tree=node(Left,Right),
     sum_tree(Left, S1),
     sum_tree(Right, S2),
     S is S1 +S2.
```

Miért jó ez nekünk???

Kényelmes

```
node(4,node(3,4))
```

alak helyett

```
node(leaf(4),node(leaf(3), leaf(4)))
```

alakot kell alkalmaznunk

első alakhoz

```
sum_tree2(Tree, S):-
      integer(Tree),
      S=Tree.
```

klózra kéne kicserélni az elsőt.

Ezt már nem lehet visszafele számolásra alkalmazni.
Ez Mercury-ban nem is megengedett.

Sicstus-ban

```
derive(x, 1).
derive(C, 0):-integer(C).
derive(U+V, DU+DV):- derive(U,DU), derive(V,DV).
…
```

Jó volna ha ugyan ezzel tudnánk integrálni is!

Megoldás:

Csomagoljuk be a constanst!

```
derive(x, 1).
derive(const(C), 0).
derive(U+V, DU+DV):- derive(U,DU), derive(V,DV).
…
```

Így viszont emberi szemmel nehezebben olvashatóak a kifejezések:
`x+3` helyett **`x+const(3)`**

További alkalmazások a megkülönböztetett uniókra:

Enumeráció szerű:

```
:- type fruit
     --->
     ;    apple
     ;    orange
     ;    banana.
```

Rekord szerű:

```
:- type employee
     ---> employee(
               name ::string,
               age  ::int,
               department ::string
).
```

Polymorf

```
:- type pair(T1,T2)
     ---> T1 - T2.

first(P::pair(string,integer),F::string) :- P = First - Sec,
                                            F = First.
```

Ekvivalencia típus

Egyszerű típus szinonima

```
:- type money==int.
```

Egy adott szerkezet megvalósítását írja le:

```
:- type assoc_list(KeyType, ValueType)
          == list(pair(KeyType),ValueType)
```

Nem lehet rekurzív!

Absztrakt típusok

A típus implementációja hiányzik. Modulok interface részében használatosak

```
:- type t1.
:- type t2(T1,T2).
```

Módok

A behelyettesítettségi állapottól függően fogalmazhatjuk meg a klózokat.

`in`: behelyettesített változót várunk
`out`: szabad változót várunk, és a klóz helyettesíti be tömör kifejezéssé.

Behelyettesítettség

Típus fa:

„vagy-csomópont”: típusokat reprezentálnak
„és-csomópontok”: konstruktorokat reprezentálnak

Behelyettesítettségi fa:

A „vagy-csomópontokhoz” rendel behelyettesítettségi információt.
`free` – Szabad
`ground` - Tömör.

A felhasználó neveket deklarálhat az egyes behelyettesítettségi fáknak

```
:- inst listskel == bound([]; [free|listskel]).
```

[A,B] – Kielégíti
[H|T] – nem elégíti ki, mert a váz csak egy része ismert.
[A,2] – nem elégíti ki, mert van egy tömör elem
[A,A] – nem elégíti ki, mert a két elem nem független egymástól

A fenti kifejezéssel ekvivalens a **bound/1** szintaktikai édesítőszere:

```
:- inst listskel ---> [] ; [free|listskel].
```

Másik példa, opcionális típus behelyettesíttettségi fája:

```
:- inst maybeskel == bound(no;yes(ground)).
```

vagy

```
:- inst maybeskel ---> no;yes(ground).
```

Parametrizált behelyettesíttettségi fák:

```
:- inst listskel(Inst) = bound([];[Inst|listskel(Inst)]).

:- inst maybeskel(Inst) ---> no; yes(Inst).
```

Módok általánosabban

Behelyettesítettségi állapot megváltozását írják le a klóz végrehajtása alatt.

```
:- mode m == inst1 >> inst2.
```

Az beépített alapmódok móddeklarációja ezek után:

```
:- mode in == ground >> ground.
:- mode out == free >> ground.
```

Módok átnevezése:

```
:- mode (+) == in.
:- mode (-) == out.
```

Parametrizált módok:

```
:- mode in(Inst) == Inst >> Inst.
:- mode out(Inst) == free >> Inst.
```

Predikatum és Függvénymód deklaráció:

```
:- mode out_listskel == free >> listskel.
:- mode in_listskel == listskel >> listskel.

:- func length(list(T)) = int
:- mode length(in_listskel)=out
:- mode length(out_listskel)=in

:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out).
:- mode append(out, out, in).
```

Mind a függvényeknek mind a predikátumoknak több módja is lehet

A **`length`** függvény esetében használhatjuk a standard library parametrikus **`in/1`**, és **`out/1`** módját

```
:- func length(list(T)) = int
:- mode length(in(listskel))=out
:- mode length(out(listskel))=in
```

Típusosztályok

Olyan típusok halmaza melyre biztosított, hogy adott függvények és predikátumok meghívhatóak, alkalmazhatóak.

```
:- typeclass point(T) where [
 pred coords(T, float, float),
 mode coords(in, out, out),

 func translate(T, float, float) = T
].
```

Típust a típusosztályba az **instance** deklarációval vehetünk fel.

```
:- type coordinate
     --->coordinate(
          float,
          float
     ).

:- instance point(coordinate) where[
     Pred(coords/3) is coordinate_coords,
     Func(translate/3) is coordinate_translate

].
```

```
:- pred coordinate_coords(coordinate, float, float).
:- mode coordinate_coords(in, out, out).

coordinate_coords(coordinate(X,Y), X, Y).


:- func coordinate_translate(coordinate, float, float)=
coordinate.

coordinate_translate(coordinate(X,Y), Dx, Dy)=
                                  coordinate(X+Dx,Y+Dy).
```

Hasonlóan felvehetjük a színes pont típust is a pont típusosztályba

```
:- type rgb ---> rgb(int,int,int).

:- type coloured_coordinate
     --->coordinate(
          float,
          float,
          rgb
     ).

:- instance point(coloured_coordinate) where[
     Pred(coords/3) is coloured_coordinate_coords,
     Func(translate/3) is coloured_coordinate_translate

].
```

```
:- pred coloured_coordinate_coords(coloured_coordinate, float,
float).
:- mode coloured_coordinate_coords(in, out, out).

coloured_coordinate_coords(coloured_coordinate(X,Y,_), X, Y).

:- func coloured_coordinate_translate(coloured_coordinate,
float, float)= coloured_coordinate.

coloured_coordinate_translate(coloured_coordinate(X,Y,C), Dx,
Dy)= coloured_coordinate(X+Dx,Y+Dy,C).
```

Típusosztály korlát predikátumokra és függvényekre

```
:- pred distance(P1, P2, float) <= (point(P1),point(P2)).
:- mode distance(in, in, out).

distance(A, B, Distance) :-
    coords(A, Xa, Ya),
    coords(B, Xb, Yb),
    XDist is Xa-Xb,
    YDist is Ya-Yb,
    Distance = sqrt(XDist* XDist+ YDist* YDist).
```

Típusosztály korlát típusosztályra

```
 :- typeclass ring(T) where [
     func zero=(T::out),
     func one=(T::out),
     func plus(T::in, T::in)=(T::out),
     func mult(T::in, T::in)=(T::out),
     func negative(T::in)=(T::out)
 ].

:- typeclass euclidean(T) <= ring(T) where [
     func div(T::in, T::in)=(T::out),
     func mod(T::in, T::in)=(T::out)
].
```

Az euclidean osztályba sorolt típusoknak ki kell elégíteniük a gyűrűbe tartozás feltételét is.
Ismerős ?
Tulajdonképpen **öröklődésről** beszélünk.

Ciao Prolog típusnyelve

Típusok szemantikájának meghatározására is lehetőséget ad.

Tulajdonságok (Properties)

Speciális predikátumok, melyek eleget tesznek annak a feltételnek, hogy a behelyettesítettségi állapotot nem változtatják meg.

```
:- prop intlist(L).

intlist([]).
intlist([X|R]):-int(X),intlist(R).

sumlist([],0).
sumlist([X|R],S):- sumlist(R,PS), S is PS+X.

:- calls sumlist(L,N) : intlist(L).
```

A hívások során elvégzi az L argumentumra az **`intlist/1`** ellenőrzést

Adott az alábbi predikátum:

```
string_concat([],L,L).
string_concat([X|Xs],L,[X|NL]):- string_concat(Xs,L,NL).
```

Mely ASCII kódban ábrázolt karakterek listájából álló stringet konkatenál.

```
:-success string_concat(A,B,C) =>
                              (compatible_with_intlist(A),
                               compatible_with_intlist(B),
                               compatible_with_intlist(C)).

:-prop compatible_with_intlist/1.

compatible_with_intlist(X) :- var(X).
compatible_with_intlist(X) :-
                          nonvar(X),
                          compatible_with_intlist_aux(X).

compatible_with_intlist_aux([]).
compatible_with_intlist_aux([X|T]) :-
                          int_compat(X),
                          compatible_with_intlist_aux(T).

int_compat(X) :- var(X).
int_compat(X) :- nonvar(X), integer(X).
```

Láthatjuk, hogy a **compat** előtag azt jeleni, hogy megvizsgájuk, hogy az argumentum változó e, ha nem változó, akkor valamilyen más tulajdonságot vizsgálunk az argumentumra (pl.: **intlist/1**)

Erre használhatjuk a standard könyvtár ilyen nevű tulajdonságát:

```
:-success string_concat(A,B,C) =>
                                (compat(intlist(A)),
                                 compat(intlist(B)),
                                 compat(intlist(C))).
```

Felhasznált irodalom:

- The Mercury Language Reference Manual
- The Ciao Prolog System - Reference Manual
- Benkő Tamás: A Mercury nagyhatékonyságú LP megvalósítás