A PARLOG párhuzamos logikai programozási nyelv

Válogatott fejezetek a logikai programozásból 2003.

Ludányi Zoltán (quad@cs.bme.hu)

(http://www.cs.bme.hu/~quad/parlog.pdf)

A Parlog jellemzői

- Független ÉS-párhuzamosság, folyam ÉS-párhuzamosság.
- Explicit (és implicit) párhuzamosság.
- Don’t-care (commited choice) nemdetermizmus.
- Őrfeltételek. (Dijkstra)
- Móddeklarációk.

Az eredeti Parlog a változókat és a predikátumokat kis-, de a strukutúrákat nagybetűvel jelölte, például „`p(S(x))`”. Mivel a legtöbb megvalósítás (például az itt használt PLP Parlog) nem ezt, hanem a Prolognál megismert szintaxist alkalmazza, ezért mi sem követjük az eredeti formát.

Vezérlés

Egy hívás következménye lehet:

- siker,
- meghiúsulás,
- felfüggesztődés. A felfüggesztődés a SICStusban megismert blokkolódáshoz hasonló állapot. (A blokkolás is a korutin-szervezés egy eszköze.) Az őrfeltételeknél lesz szerepe.

Egy konjukció:

- Egy konjunkció sikerül, ha minden tagja sikerül.
- Egy konjunkicó felfüggesztődik, ha egy tagja felfüggesztődik.
- Egy konjunkció meghiúsul, ha bármely tagja meghiúsul.

Móddeklaráció

- Az ismertetett `termelő/2` – `fogyasztó/1` példában (plp/18) a „`:- block fogyasztó(-).`” deklaráció azért szükséges, hogy a logikai változó a híváskor ne egyesüljön azonnal az `F = [N1|F1]` listával.
- Móddeklarácók nélkül egy kód több irányba is használható, pedig lehet, hogy csak az egyik irányban működik. (Jobb esetben egy irányban optimális.)
- A Prologban nincs lehetőség móddeklarációkra, azokat legfeljebb kommentként írhatjuk oda (például `mode/1`.), aminek viszont nincs hatása a futásra. Helyette a `var/1` beépített eljárás segítségével lehet elágazni. (Ha sokat használjuk, akkor ez a megoldás csúnya.)
- Parlogban móddeklarációt a `mode/1` deklarációval érhetünk el. Hatása a `block/1`-hez hasonló.

A móddeklaráció szintaktikája

Prolog direktíva	Prolog `block/1`	Parlog
`mode p(Arg1,...,ArgN).`	`:-block p(Arg1,...,ArgN).`	`mode p(Arg1,...,ArgN).`
`Argi = [ in | out ]`	`Argi = [ - | ? ]`	`Argi = [ ? | ^ ]`

- A `?` és `^` jelek elé egy névkonstans is tehető.
- A kimeneti változónak a meghívás pillanatában egy behelyettesítetlen változónak kell lennie. Bemeneti változókra nem az egyesítés, hanem az illesztés primitív fog meghívódni.
- Például folyam ÉS-párhuzamosság esetén a móddeklarációkból állapítható meg a folyamatok közötti viszony.

```
(folyam.par)
mode termelő(mettol?,folyam_ki^). mode fogyasztó(folyam_be?).
        <- termelő(1,F), fogyasztó(F).
```

Illesztés

- A Parlog nyelv az egyesítés (`=/2`) és az azonosság (`==/2`) mellé bevezeti az illesztést is (`<=/2`).
- Az illesztés tulajdonképpen az egyirányú egyesítés (one-way unification), móddeklarációja: `mode <=(?,?)`. Minden felfüggesztődés erre a hívásra vezethető vissza.

`X` illesztése `Kif`-re: `Kif <= X.`

1. A fenti hívás lefutása után `X` és `Kif` szintaktikusan azonosak lesznek, vagyis az illesztés után a `Kif == X` hívás sikerrel fut le.
2. Ennek eléréshez azonban csak `Kif`-ben történhetnek egyesítések.
3. Ha az azonosság eléréséhez `X`-ben is le kellene kötni változókat, akkor a hívás felfüggesztődik.

Példa (`X` illesztése `Kif`-re):

```
Kif <= X.
```

Csak `Kif`-ben történhetnek egyesítések. A jelet (szerintem) prakitkus úgy tekinteni, hogy az `X`-ben lévő értékeket bemásoljuk `Kif`-be. (A nyíl a bemásolás irányát mutatja.)

Kif	X	Jelentése
Kif	X	Kif = X
a	X	(X==a)
Kif	a	Kif = a
[1]	a	fail
s(A,B,C)	s(1,2,3)	A=1 B=2 C=3
s(A,B,3)	s(1,C,D)	(A=1 B=C D==3)
s([1])	s([1,2])	fail

- A zárójelbe tett részek most a felfüggesztett hívásokat jelölik. Például `(A==3)` azt jelenti, hogy a hívás mindaddig felfüggesztődik, amíg `A` be nem helyettesítődik a `3` értékre. Ha valóban erre az értékre helyettesítődik be, akkor sikerrel lefut; ha más értékre, akkor meghiúsul.
- Az illesztés tehát a `block` deklarációval rokon, mert egy változó behelyettesítődéséig felfüggeszti a hívást.
- Az illesztés azonban a blokkolásnál erősebb, mert a blokkolás csak a legkülső szinten figyeli a változót (`A = [_]` esetén a blokkolás felenged). Ezzel szemben az illesztés egy „mélyblokkolás”, ami mindaddig felfüggesztődik, amíg akár csak egy részstruktúra illesztése is fel van függesztődve.
- A mélyblokkolás miatt egyszerű blokkolással az illesztés nem is valósítható meg, mert blokkolásnál a változó legkülső szintű behelyettesítése után megtörténik az egyesítés. Illesztésnél semmilyen behelyettesítés nem történik addig, amíg a teljes illesztés sikerrel le nem tud futni.

- Az illesztés más nyelvekben is ismert fogalom. Például a CHR-ben is ilyet alkalmaztunk a klózok illesztésénél. (`neg1@ neg(0,X) <=> X = 1.`)

Illesztés megvalósítása Prologban

```
:- op(700, xfx, <=).
<=(General, Specific) :- sumsumes(General, Specific).

subsumes(General, Specific) :-
        subsumes_chk(General, Specific),
        General = Specific.

subsumes_chk(General, Specific) :-
        \+ \+ ( numbervars(Specific, 0, _), General = Specific ).
```

Egy példa (unio.par)

```
mode unio(halmaz1?, halmaz2?, unio^).
unio([X|H1], H2, [X|H3]) <- unio(H1, H2, H3).
unio(H1, [X|H2], [X|H3]) <- unio(H1, H2, H3).
unio([], H2, H2).
unio(H1, [], H1).

<- load(between), load(unio), compile.
<- U: unio(H1,H2,U), lazy_between(1,5,H1), lazy_between(6,10,H2).
U = [6,1,2,7,3,8,9,4,5,10]
succeeded

<- U::: unio([1,2|L1],[3,4|L2], U).
                ( U = [1,2,3,4|_0005722F] )
```

A commited choice nemdeterminizmus

- (Őrfeltételek hiányában) a klózok között kizárólag a bemeneti változók illesztése alapján tudunk dönteni. Ha valamelyik klózra az illesztés sikeres, akkor elkötelezzük magunkat mellette függetlenül a többi klóztól. A sikeres bemeneti illesztésű klózt *kijelöltnek* (candidate) hívjuk

Az illesztés jelentősége

- Azt már láttuk a blokkolásnál is láttuk, hogy a bemeneti változókban egyesítést nem használhatunk.
- A blokkolás az `[1], [X]` és `[X|L]` eseteket nem különbözteti meg.
- Ami azonban a legfontosabb: a kimeneti változók egyesítése mindaddig nem történhet meg, amíg a bemeneti változók illesztése sikerrel le nem fut. (Hiszen lehet, hogy az illesztés egy későbbi behelyettesítés miatt meghiúsul!)

Klózok közötti választás

- A klózok között tulajdonképpen egy committed VAGY-párhuzamos keresés zajlik.
- A bemeneti argumentumok sikeres illesztésének feltételét egy adott klóz esetén a klóz *bemeneti korlátjának* (input constraint) hívjuk.
- Híváskor tehát három lehetőség adott:

  1. Az argumentumok valamely klózban kielégítik a bemeneti korlátot. Sikeres illesztés történik, az adott klóz mellett elkötelezzük magunkat.

  2. Az argumentumok egyik klózzal sem elégíthetik ki a bemeneti korlátot, a hívás meghiúsul.

  3. Az argumentumok csak a bemeneti korlát (az illesztés) megsértésével helyettesítődhetnének be. A hívás felfüggesztődik, amíg az egyik klózban (egy máshol történt egyesítés miatt) sikeres illesztés nem történik.

Őrök

- Néha csak a bementi illesztésnél bonyolultabb (pl. aritmetikai) feltételt tudunk megfogalmazni a klózok közötti választásra. Ilyenkor őröket alkalmazhatunk.
- Az őrfeltétel szintén egy konjunkció, amit a commit operátor (`':/2'`) zár.
- Egy predikátum szintaxisa (az argumentumok jelölése nélkül) az őrfeltétellel („`g1, ..., gK`”) kiegészítve:

```
p <- g1, ..., gK : p1, ..., pM.
```

- Az őr hívásainak a bemeneti illesztéssel együtt sikeresen le kell futniuk ahhoz, az adott klóz mellett kötelezzük el magunkat.
- A commit operátor a vágóhoz hasonló, de itt a klózok között párhuzamosan keresünk, így köztük nem a sorrendiség, hanem az időbeliség dönt a vágás pillanatában.

Példa aritmetikai őrre: keresés rendezett fában (`bfaban.par`)

```
mode bfaban(elem?, ebben_a_bfaban?).
bfaban(Elem, bfa(Elem,_,_)).
bfaban(Elem, bfa(Kulcs,BalFa,_)) <-
        Elem < Kulcs
        :
        bfaban(Elem, BalFa).
bfaban(Elem, bfa(Kulcs,_,JobbFa)) <-
        Kulcs < Elem
        :
        bfaban(Elem, JobbFa).
```

- Ha nincs őrfeltétel, akkor a vágásnak megfelelő elkötelezés a már megismert módon a bemeneti illesztés után azonnal megtörténik. Ekkor az őrfeltétel elhagyható, vagyis a két alábbi forma azonos.

```
p <- true : p1, ..., pM.      p <- p1, ..., pM.
```

- Sőt, a bemeneti korlát explicit kiírásával a bemeneti korlát fogalma eldobható, és a klózok közötti választás csak az őrfeltételre alapozható. Ha minden hívásban kifejtjük a bemeneti formát, akkor ún. *standard formához* (standard form) jutunk. (Ezt az átírást a fordító a móddeklarációk alapján meg is teszi. Az átírás után a móddeklarációkra már nincs is szükség.) A fentiek miatt mostantól a bemeneti korlátot mi is az őr részének tekintjük.

- Példa:

```
mode p(bemenet?).      p([X|L]) <- őr : törzs.
```

Ekvivalens formában (Kernel Parlog):

```
p(Arg) <- [X|L] <= Arg, őr : törzs.
```

- Korábban kiemeltük, hogy „a kimeneti változók egyesítése mindaddig nem történhet meg, amíg a bemeneti változók illesztése sikerrel le nem fut”. Az őrfeltétek bevezetése miatt ezt az állítást újra kell fogalmaznunk: a kimeneti változók egyesítése mindaddig nem látszódhat, amíg az őrfeltétel sikeres lefutása után el nem köteleztük magunkat az adott klóz mellett.
- Miért? Ismét csak azért, mert a felfüggesztett hívások meghiúsulhatnak, és az így nyert hamis egyesítések más felfüggesztett hívásokat is elindíthatnak. A Parlogban azonban nincs visszalépés, és egy ilyen inkonzisztens állapot legfeljebb meghiúsuláshoz vezethet.
- A régebben használt ‘megtörténhet’ szót itt azért kell a ‘látszódhat’ szóra cserélni, mert a bemeneti illesztés (többnyire) nincs hatással a kimeneti változókra. Az őrfeltételekben azonban gyakran megtörténik egyes kimeneti változók behelyettesítése is, ami az előző pontban említettek miatt nem látszódhat a többi hívás számára.

Klózok közötti választás az őrök tükrében

- Kijelölt (candidate) klóz: olyan klóz, amiben az őrfeltétel sikeresen lefutott. A sikeres lefutás után az adott klóz mellett kötelezzük el magunkat (commit).
- Nemjelölt (non-candidate) klóz: olyan klóz, amiben az őrfeltétel meghiúsult.
- Felfüggesztett klóz: olyan klóz, amiben legalább az egyik hívás felfüggesztődött, és egyik sem hiúsult meg.

Az őr hatása a hívás sikerére

- Siker esetén megtörténik a redukció és a kimeneti változók egyesítése.
- Ha minden klóz nemjelöltté válik, akkor a hívás is meghiúsul.
- Ha vannak felfüggesztett klózok, de még nincs kijelölt, akkor maga a hívás is felfüggesztődik.

A hívás lefutásának következményei

- Ha egy konjunkció bármely tagja meghiúsul az őrben (de általában is), akkor a többi hívást is le kell állítani.
- Ha egy klóz mellett elköteleztük magunkat, akkor a többi őrt le kell állítani.

Megjegyzés

Az őr is ismerős a CHR-ből, például:

```
neg2@   neg(X,Y) \ vagy(X,Y,Z) <=> Z=1.
```

Sorosítás

Soros konjunkciók (sequential conjunctions)

- A Parlog a párhuzamos konjunkció (`','/2`) mellé bevezeti a soros konjukciót (`'&'/2`) is.
- Parlogban definíció szerint a két konjuktív operátor azonos szintű precedenciával bír, de a legtöbb megvalósítás ezt nem követi, és a párhuzamos konjunkciót (`op(1000,xfy,',')` magasabb precedenciával bír, mint a soros (`op(1025,xfy,'&')`. Tehát ha nem használunk zárójeleket, akkor párhuzamos folyamatok fognak egymás után végrehajtódni.
- Zárójeleket lehet használni, így tetszőlegesen bonyolult párhuzamossági ágak is kialakíthatóak. (Bármi, ami a *fork-join* párral megvalósítható.)
- Többnyire csak a mellékhatásos eljárásokhoz van rá szükség.

Soros konjunkció teszt

A kérdés: kell-e sorosítani, és ha igen, akkor miért?

```
mode writelist(kiirando_lista?).
writelist([X|L]) <-
        write(X) & nl & writelist(L).
writelist([]).

mode between(from?, to?, from_to_list^).
between(N, M, [N|L]) <-
        N < M :
        N1 is N + 1 & between(N1, M, L).
between(N, N, [N]).
```

Soros választás klózok között

- Parlogban praktikus a klózokat nem különálló egységnek, hanem egy struktúrának tekinteni. A klózokat elválasztó `'./2'` operátor így többletjelentést kap (az utolsó nem): az általa elválasztott klózsorozat is egy konjunkciónak tekinthető. Például a két alábbi forma azonos:

```
p(1). p(2). p(3).
.(p(1),.(p(2),p(3))).
```

- Egy ilyen konjunkcióban a kijelölt klóz keresése párhuzamosan történik.
- Azonban itt is lehetőség van a sorosításra, ha `'./2'` helyett `';/2'`-t használunk. Használatakor a `';'` után álló klózok mindaddig nem lesznek megvizsgálva, amíg az előtte állók meg nem hiúsultak. (`'./2'` precedenciája kisebb, mint `';/2'`-é ahogy azt a konjunkciók esetében már megszoktuk.)

```
p(1). p(2); p(3).
;(.(p(1),p(2)),p(3))).
```

- Mi `'.'/2`-t *párhuzamos választásnak*, `';'/1`-t *soros választásnak* fogjuk hívni, bevett terminológia nincs.
- A klózok a Parlog nyelvének definíciója szerint zárójelezhetőek is, a gyakorlatban ez nem megengedett.
- A soros konjunkció és a negálás műveletével a soros választás megvalósítható. Például az alábbi két forma ekvivalens, de nem egyformán hatékony (az argumentumok most lényegtelenek, ezért elhagyjuk őket).
- (A tagadásra Parlogban a `'~'/1` van definiálva, de a PLP a `'not'/1`-et használja.)

```
mode p.
p <- g1 : p1;
p <- g2 : p2;
p <- p3;
```

```
mode p.
p <- g1 : p1.
p <- not(g1) & g2 : p2.
p <- not(g1) & not(g2) : p3;
```

- A soros választásra akkor lehet szükség, ha nem akarjuk az erőforrásokat feleslegesen pazarolni. A `';'/1` nem csupán szintaktikus édesítőszer, éppen ezért nem érdemes átírni a fent bemutatott módon. Viszont ne használjuk, ha nem vagyunk benne biztosak, hogy a döntéshez szükséges változók rendelkezésre fognak állni.

Példa: keresés rendezetlen fában (`faban.par`)

```
mode faban(elem?, ebben_a_faban?).

faban(Elem, fa(Elem,_,_))
;
faban(Elem, fa(_,BalFa,_)) <-
        faban(Elem, BalFa).
faban(Elem, fa(_,_,JobbFa)) <-
        faban(Elem, JobbFa).
```

Granularitás

- Ha túl sok soros konjunkciót kell alkalmazni, akkor programozzunk inkább Prologban. Ha túl keveset, akkor a processzort túlterheljük, mert állandóan váltogatni kell a szálak között. Ha nem váltogatna, akkor nem lenne biztos, hogy a fogyasztó folyamatok tudnak is fogyasztani. (Lásd fair tulajdonság.)
- Ha túl sok soros választást alkalmazunk, akkor ismét lassulhat a program; lehet, hogy a feltételek alapján a két klóz párhuzamosan is futhat. Ha nem használjuk, akkor lehet, hogy feleslegesen kerülnek klózok megvizsgálásra – lásd a példát.
- Steve Gregory Parlogja (az eredeti) felajánlja a programozónak, hogy az architektúrától függő végrehajtást írjon elő. Ennek megfelelően az `'and'/2` operátor vagy `','/2`, vagy `'&'/2` operátorrá válik fordításkor, hasonlóan a `'..'/2` `'.'/2` vagy `';'/2`-vé.
- A PLP-ben ilyen lehetőség nincs.

Programozás Parlogban

Korábban már említettük, hogy a commited choice nemdeterminisztikus nyelveknél a keresés az őr segítségével lehetséges, de nem egyszerű, drága és csak egy megoldása lehet.

Példa: két halmaz egy közös elemének a megtalálása.

Prolgban (`kereses.pl`)

```
:- mode eleme(in, in).
:- mode eleme(out, in).
eleme(E, [E|_]).
eleme(E, [_|L]) :-
        eleme(E, L).
```

```
:- mode közös_eleme(in, in, in).
:- mode közös_eleme(out, in, in).
közös_eleme(K, L1, L2) :-
        eleme(K, L1),
        eleme(K, L2).
```

Parlogban (`kereses.par`)

- keresést írni nem lehet, mert nincs választási pont,
- azt explicit bele kell kódolni az őrbe,
- a keresésre szolgáló hívások egy része ellenőrzéssé alakítható.

```
mode eleme(elem?, lista?).
eleme(E, [F|_]) <-
        E == F
        :
        true.
eleme(E, [F|L]) <-
        E =\= F
        :
        eleme(E, L).
```

```
mode kozos_eleme(elem^, lista1?, lista2?).
kozos_eleme(K, [E|L1], L2) <-
        eleme_e(E, L2)
        :
        K = E.
kozos_eleme(K, [_|L1], L2) <-
        kozos_eleme(K, L1, L2)
        :
        true.
```

Hogyan NE programozzunk Parlogban

A fent bemutatott módon, vagy csak a legszükségesebb esetben. (Ha keresésre van szükség, akkor érdemes megfontolni az áttérést valamilyen don't-know nemdeterminizmust támogató nyelvre.)

Miért?

- Az őr hatása nem látszódhat mindaddig, amíg sikerrel le nem futott, és el nem köteleztük magunkat az adott klóz mellett, így a benne szereplő hívásokat elkülönítve kell kezelni.
- Még rosszabb a helyzet, ha több szinten is mély hívásokat végzünk az őrben.
- Keresés során gyakran alkalmazunk rekurziót (pl.`eleme/2`), aminél biztos, hogy a fenti eset áll elő.

Példa rekurzív őrre (`test.pl, test.par`)

```
mode append1(l1?, l2?, l12^).
append1([H|L1], L2, [H|L3]) <-
        append1(L1, L2, L3)
        :
        true.
append1([], L, L).
```

```
mode append2(l1?, lt2?, l12^).
append2([H|L1], L2, [H|L3]) <-
     append2(L1, L2, L3).


append2([], L, L).
```

```
<- load(test) & (optimize on) & compile.
# between.par loaded
# test.par loaded
succeeded
```

```
<- test(1).
Error 4, Heap Space Full,
Trying ' seqand'/6
```

```
<- test(2).
3840
succeeded
```

Hogyan programozzunk hatékonyan keresést (`kereses.par`)

- A nemdeterminisztikus keresést próbáljuk meg determinizálni.
- A választási pontban szükséges információkat „mentsük el” egy újabb argumentumba, valamint ügyeljünk rá, hogy az első választási pont kimerítése nem meghiúsulással, járjon, hanem a további lehetőségek ellenőrzésével (az új argumentumból kivehető adatok alapján).
- A fenti módszer SML-ből ismerős lehet, bár ott ennél kényelmesebb, de kevésbé hatékony módszerek is rendelkezésre állnak, például a megoldások listába gyűjtése esetén `@`.

  Példa a következő oldalon!

```
mode kozos_eleme2(kozos_elem^, lista1?, lista2?).
kozos_eleme2(K, L1, L2) <-
        L1 =\= []
        :
        kozos_eleme3(K, L1, L2, L1).

mode kozos_eleme3(kozos_elem^, lista1?, lista2?, lista1teljes?).
kozos_eleme3(K, [E1|_], [E2|_], _) <-
        E1 == E2 :
        K = E1.
kozos_eleme3(K, [E1|L1], [E2|L2], L1T) <-
        E1 =\= E2 :
        kozos_eleme3(K, L1, [E2|L2], L1T).
kozos_eleme3(K, [], [_|L2], L1T) <-
        kozos_eleme3(K, L1T, L2, L1T).
```