Párhuzamos logikai programozás

Válogatott fejezetek a logikai programozásból 2003.

Ludányi Zoltán (quad@cs.bme.hu)

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

Párhuzamosság

Párhuzamosság

```
f_x = f(x); g_y = g(y);        hf_x = h(f_x);
```

Két kérdés: - mi futhat párhuzamosan?

- a párhuzamosságot ki vezérelje?

Párhuzamosság vezérlése

- *Implicit* párhuzamosság: a programozó számára láthatatlan, fordítási időben automatikusan dől el a függőségek alapján.
- *Explicit* párhuzamosság: a programozó vezérel.

Párhuzamosság funkcionális nyelvekben

SML nyelvű példa:

```
fun f(x,y) = g( h1(x), h2(y), h3(h1(x)) );
fun g(x,y,z) = x@y@z;
```

Független párhuzamosság (independent parallelism)

- Csak az argumentumok kiértékelése konkurens.
- Az argumentumok mindig függetlenek, mivel nincsenek logikai változók, így a megvalósítása könnyű.
- Szokás *korlátozott* (restricted) párhuzamosságnak is hívni.

SML nyelvű példa:

```
fun f(x,y) = g( h1(x), h2(y), h3(h1(x)) );
fun g(x,y,z) = x@y@z;
```

Függő párhuzamosság (dependent parallelism)

- Egy kifejezés és az argumentumainak konkurens kiértékelése.
- A fordítótól elvárjuk, hogy a fentiekhez hasonló függéseket felismerje, és azokat hatékonyan kezelje. (A `h3(h1(x))` argumentum kiértékelésekor `h1(x)` ne értékelődjön ki újra – vagy csak ott.)

```
fun f(x,y) = g( h1(x), h2(y), h3(h1(x)) );
```

Átírva:

```
fun f(x,y) =     let     val h1_x = h1(x)
                         val h2_y = h2(y)
                         val h13_x = h3(h1_x)
                 in      g(h1_x, h2_y, h13_x)
                 end;
```

Párhuzamosítás esetén itt `h1_x`-et `h13_x` előtt kell kiszámítani, mivel függ tőle, tehát lokális változók esetén már itt is jelentkeznek az imperatív nyelveknél megismert egyes problémák!

Ilyen kérdés például az, hogy mely műveleteket célszerű párhuzamosítani. (Az architektúrától függ.) Ugyanakkor a változók nem frissíthetőek, ami nyilván egyszerűsíti a párhuzamosítást.

Párhuzamosság logikai nyelvekben

Logikai nyelveken argumentum kiértékelés nincs, helyette az átalakított SML kódhoz hasonlóan minden értéket külön kiszámítunk.

Az SML példa Prologban:

```
fun f(x,y) =
let     val h1_x = h1(x)
        val h2_y = h2(y)
        val h13_x = h3(h1_x)
in      g(h1_x, h2_y, h13_x)
end;
```

```
f(X, Y, Z) :-
        h1(X, H1_X),
        h2(Y, H2_Y),
        h3(H1_X, H13_X),
        g(H1_X, H2_Y, H13_X, Z)
        .
```

Párhuzamosság konjunkciókban

```
f(X, Y, Z) :-    h1(X, H1_X), h2(Y, H2_Y), h3(H1_X, H13_X),
                 g(H1_X, H2_Y, H13_X, Z).
```

- A Horn klózok a logikára alapulnak, a predikátum a törzsben szereplő tagok sorrendjétől független (kellene, hogy legyen). Például tisztán ellenőrzéskor lehet is, ha nem használunk vágót.
- A valóságban mindig van egy *végrehajtási modell* (Prolognál a soros redukció, de létezik az ún. *táblázás* módszere is), aminek tükrében programozunk. Erre azért van szükség, mert általában nem ellenőrzünk, hanem generálunk. A végrehajtási modell így speciális programozási stílust követelhet meg. Például Prologban a közvetlen balrekurzió (`p :- p, q.`) nem praktikus.
- A konjunkció sorrendfüggőségének megkerülésére szolgálhat például a `block` deklaráció. (Persze a blokkolás ennél többre jó!) Adott be- és kimeneti változók esetén egyszerű, többmódú predikátumoknál vizsgálattal elágazhatunk. (Lásd NHLP.)

Párhuzamosság diszjunkciókban

```
p(X) :- ( X = 1 ; X = 2 ).
p(X) :- X = valami_más.
```

- A keresési fa ágainak párhuzamos bejárása szintén kínálja a párhuzamosítást.
- A végrehajtás szempontjából a klózok sorrendje legyen lényegtelen. (Matematikaibb szemlélet, szintén a Horn klózok logikai oldala.)

Nemdeterminizmus

‘Nemismert’ nemdeterminizmus (don't-know non-determinism)

- A választás pillanatában nem tudjuk eldönteni, hogy melyik ág vezet majd el a jó eredményre. Például: `bűvös_csiga/2`.
- Akkor kell alkalmazni, ha valóban keresésre van szükség, ugyanis az adott pillanatban a választási pontok közötti döntés nehéz vagy lehetetlen.
- Amennyiben megengedjük, és ennek mentén párhuzamosítunk, akkor az egyes változók a különböző keresési ágakon más és más behelyettesítést kaphatnak.
- Ennek a problémának a megoldásához a klasszikus egyszeri értékadás (single assignment) nem elégséges.

- A különböző keresési ágak mentén egy változó különböző értékekkel helyettesítődhet be, így ha a keresési fát párhuzamosan járjuk be, akkor be kell vezetni a *többszörös kötés* (multiple binding) fogalmát. (Egy változó különböző példányai különböző értékekkel helyettesíthetőek be.)
- A keresési ágak mentén ennek megfelelően kétféle kötés alakulhat ki (lásd WAM!):

  Feltétel nélküli (unconditional) kötés akkor történik, ha az adott változónak a keresési fában csak egy példánya van. (Például a hívásban, amiben ő szerepel nincs választási pont.) Ilyenkor az adott változó egyszeri értékadású.

  F*eltételes* (conditional) kötés ezzel szemben akkor alakul ki, ha a változó a különböző keresési ágak mentén különböző értékeket kaphat. Ilyenkor többszöri értékadású változóval kell dolgozni.

egyszeri értékadás	feltétel nélküli kötés
többszöri értékadás	feltételes kötés

‘Mindegy’ nemdeterminizmus (don't-care non-determinism)

(unió.pl)

```
:- block unió(-, -, ?).
unió(A, B, AB) :-
(       var(B) ->
        (       A == [] -> AB = B
        ;       A = [E|A1], AB = [E|AB1],
                unió(A1, B, AB1)
        )
;       (       B == [] -> AB = A
        ;       B = [E|B1], AB = [E|AB1],
                unió(A, B1, AB1)
        )
).
```

‘Mindegy’ nemdeterminizmus (don't-care non-determinism)

- Szokás *elkötelezett választású* (commited choice) nemdeterminizmusnak is hívni, mert itt a választás után el kell kötelezni magunkat egy adott ág mellett. Szintaxis: `p :- őr1 : törzs1. p :- őr2 : törzs2.` ...
- Akkor van értelme, ha a választás szabad, vagyis valóban mindegy, hogy melyik ág mellett kötelezzük el magunkat.
- A döntés pillanatában alkalmazott `commit` művelet olyan, mint egy vágás (`!/0`), de itt minden klózba kötelesek vagyunk valahova vágót tenni, valamint a klózok között a (kódbeli) sorrendiség lényegtelen.
- A döntést persze lehet más hívásokra is alapozni, a `commit`-ot megelőző részt *őrnek* hívják. (Dijkstra)
- A don’t-know nemderminizmussal szemben itt elég a klasszikus egyszeri értékadású logikai változókban gondolkodni, azonban így a hívások között függőségek léphetnek fel.

Párhuzamossági alaptípusok

A főbb alaptípusok

- Egyesítési párhuzamosság
- ÉS-párhuzamosság
- VAGY-párhuzamosság
- Kombinált ÉS/VAGY-párhuzamosság

Egyesítési párhuzamosság (unification parallelism)

- Tetszőleges egyesítés párhuzamosítása a részstruktúrák mentén.
- A hívás és a klóz fej argumentumainak párhuzamos egyesítése. (Persze a hívás is egy struktúra.)

ÉS-párhuzamosság (AND-parallelism)

ÉS-párhuzamosság (AND-parallelism)

Párhuzamosság egy keresési ágon belül.

- Független ÉS-párhuzamosság: egy konjunkció független hívásainak konkurens kiértékelése. Például a hívások nem osztoznak (behelyettesítetlen) változón.
- Függő ÉS-párhuzamosság: egy konjunkció tetszőleges (nem feltétlenül független) hívásainak konkurens kiértékelése.
- Folyam ÉS-párhuzamosság (stream AND-parallelism): a függő ÉS-párhuzamosság speciális esete. A párhuzamosan futó folyamatok között egy *folyamot* hozunk létre, amin keresztül az információ áramlik.

Független ÉS-párhuzamosság

Megköveteli a változók függetlenségét.

Példa:

```
p(X,Y,Z) :- q1(X,X1), q2(Y,Y1), r(X1,Y1,Z).
```

- A `q1` és `q2` hívások (várhatóan) függetlenek, nem hatnak egymásra. Akár két teljesen független processzor is feldolgozhatná őket független memóriaterülettel.
- Az `r` hívás `q1`-től és `q2`-től is függ, így ő egyikükkel sem párhuzamosítható.
- Egy tipikus használat: $M \cdot v$ (mátrix – vektor szorzat). (`mxv.pl`)

```
mxv([], _, []).
mxv([Sor|Mátrix], V, [Szorzat|Maradék]) :-
        skalár_szorzat(Sor, V, 1, Szorzat),
        mxv(Mátrix, V, Maradék).
```

Függő ÉS-párhuzamosság: (nem)determinizmus

- Don’t-know nemdeterminisztikus környezetben a megvalósítás nehéz.

```
p1(X) :- ( X = 1 ; X = 2 ).
p2(X) :- ( X = 2 ; X = 1 ).
| ?- p1(X), p2(X).
```

- Determinisztikus környezetben könnyű, mivel a determinisztikus programok függvényeknek is tekinthetőek. Ekkor bátran lehet egyszeri értékadású változókkal dolgozni, mert csupán közös pointereket kell alkalmazni, és ha bármelyik változóban ütközés következne be, akkor egyszerűen meg kell hiúsulni. (Közös memóriaterület esetén zárolásra lehet szükség.)
- Don't-care nemdeterminizmus esetén nincs választás, de az őrfeltételekben lévő esetleges kereséshez bevezetésre kerül az *illesztés* fogalma.

Függő ÉS-párhuzamosság pro és contra

Determinisztikus vagy don’t-care nemdeterminisztikus környezetben:

- Előny: az inkrementális kommunikáció egyszerű, a klasszikus egyszeri értékadású változókat kell használni zárolással, ahol a behelyettesítés azonnal látszik.
- Hátrány: csak egy megoldás lehetséges, ha több megoldás kell, akkor azt listába kell gyűjteni, és ez nehézkes.

Don’t-know nemdeterminisztikus környezetben:

- Előny: maga a nemdeterminizmus.
- Hátrány(?): VAGY-párhuzamossággal kell kombinálni, utána viszont a változók a különböző ágakon függetlennek tekinthetőek.

Keresés függő ÉS-párhuzamosság mellett

A látszat ellenére keresés don't-care nemdeterminizmus esetében is lehetséges, de csak az őr segítségével, és:

- annak csak egy megoldása lehet (vágás),
- általában a döntéshez használjuk fel,
- drága!

A keresést általában nem lehet egyszerűen meghívni, azt expilcit *bele kell kódolni* a klózokba. Így pontosan annyi klóz jön létre, ahány választási pontunk van. (Nem szép, lásd `kereses.par`. Erre inkább a Parlog megismerése után térünk vissza, mert nehéz.)

Folyam ÉS-párhuzamosság

- A működésben *termelő* (producer) és *fogyasztó* (folyamatok) fedezhetőek fel. (`folyam.pl`)
- Az *inkrementális* kommunikáció a két folyamat között létrehozott *folyamon* keresztül történik.
- Hoare: CSP (Communicating Sequential Processes), például Occam, ProMeLa.

```
termelő(N, F) :-
        F = [N|F1],
        N1 is N+1,
        termelő(N1, F1).
```

```
:- block fogyasztó(-).
fogyasztó(F) :-
     F = [N|F1],
     write(N),
     fogyasztó(F1).
```

```
| ?- fogyasztó(F), termelő(1, F).
```

VAGY-párhuzamosság (OR-parallelism)

Példa:

```
p(X) :- ( X = 1 ; X = 2 ).
p(X) :- X = valami_más.
```

Mindkét klóz egyszerre kezd el futni, az elsőn belül a választási pont mentén elágazunk, vagyis három keresési ág alakul ki párhuzamosan.

- Párhuzamosság egy híváson belül, a klózok és választási pontok konkurrens kiértékelése.
- A keresési ágak párhuzamosan kerülnek bejárásra.

- A don’t-know nemdeterminizmuson alapul, így a megvalósításhoz többszörös kötés kell. (Egy logikai változó különböző ágakon különböző értékekkel lehet behelyettesítve.)
- Elméletileg tetszőleges ÉS-párhuzamos modellel kombinálható, akár a klasszikus Prologgal is, amit szokás ÉS-*soros* rendszernek is hívni.
- A megoldásokat lehet sorrendben visszaadni (de akkor ki kell várni az első végét), vagy lehet összefésülni. A vágás megőrzéséhez a klózok között meg kell tartani sorrendet.
- DFS helyett BFS-szerű, így kiküszöböli a Prolog nem-teljességét. (A végtelen ágak a többi megoldást nem érintik.)
- Ott használható jól, ahol sok választási pont van: szakértő rendszerek, természetes nyelvű feldolgozás, adatbázisok.

Egy történelmi relikvia: a pipeline elv alkalmazása

```
p(X) :- p1(X), p2(X).
```

Klasszikus Prolog végrehajtás:

1. `p1` (esetleg részlegesen) behelyettesíti `X`-et.
2. `p2` ellenőrzi a megoldást (esetleg további behelyettesítéseket végezve).

A predikátumok csak meghiúsulás után keresnek új megoldásokat.

Pipeline végrehajtás:

1. `p1` (esetleg részlegesen) behelyettesíti `X`-et.
2. `p2` ellenőrzi a megoldást (esetleg további behelyettesítéseket végezve) miközben `p1` újabb megoldásokat keres `X`-re

Mi történjen `p1` további megoldásaival?

- Buffereljük őket. (Nulla hosszú buffer esetén Prologot kapunk.)
- Minden megoldást továbbadunk `p2`-nek. Lehet, hogy `p2` után más is további ellenőrzéseket végez, és ő két majd új megoldást. (Minden új megoldásnál elágazunk (fork).)

Pro és contra

- Megtartja a szekvenciális sorrendet, ha a keresésben továbbra is megtartjuk a sorrendet, akkor a vágó itt is működhet.
- Spekulatív, vagyis lehet, hogy feleslegesen buzgó.
- A végtelen ciklusok itt is bajt okozhatnak.

Kombinált ÉS/VAGY-párhuzamosság

```
p(X) :- p1(X), p2(X).
```

Klasszikus Prolog végrehajtás:

1. `p1` (esetleg részlegesen) behelyettesíti `X`-et.
2. `p2` ellenőrzi a megoldást (esetleg további behelyettesítéseket végezve).

A predikátumok csak meghiúsulás után keresnek új megoldásokat.

Konkurrens végrehajtás:

1. `p1` és `p2` egymástól függetlenül, egyszerre keres megoldásokat.
2. A két megoldáshalmazból kiválogatjuk az azonosakat (*back unification*).

Végtelen ciklusok, hülye helyzetek lassíthatják, vagy megölhetik.

A független ÉS-párhuzamosság egy javításának tekinthető.

Kombinált ÉS/VAGY-párhuzamosság

A fenti megoldás kiváltható a `bagof/3` és a `metszet/3` hívásokkal.

(Azért `bagof/3`, és nem `findall/3`, mert az összefüggő hívások más változókon is osztozhatnak.)

Példa:

```
p(X) :- p1(X), p2(X).
```

Átírva:

```
p(X) :-  bagof(X, p1(X), XL1),
         bagof(X, p2(X), XL2),
         metszet(XL1, XL2, XL),
         member(X, XL).
```