Prolog programok párhuzamos végrehajtása

Válogatott fejezetek a logikai programozásból szeminárium,

Farkas Gábor

2005.12.05.

Fibonacci sorozat (buta rekurzív) számítása

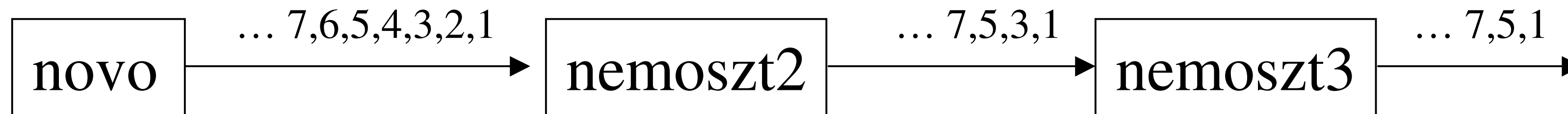
```
fib(0,1).
fib(1,1).
fib(M,N) :- [ M1 is M - 1, fib(M1, N1) ],
            [ M2 is M - 2, fib(M2, N2) ],
            N is N1 + N2.
```

A szögletes zárójelbe helyezett két cél(halmaz) láthatóan párhuzamosan végrehajtható. Hasonlóan párhuzamosítható a mátrix-szorzás, az összefésüléses rendezés, a gyorsrendezés. A két cél (változó)független ÉS kapcsolatban áll.

Termelő-fogyasztó összeállítás

```
% L az [K, K+1, ..., N ] lista.
novo(K, N, L) :-
        (   N >= K -> L = [K|L1], K1 is K+1, novo(K1, N, L1)
        ;   L = []
        ).
% Az L1 egészlista K-val nem osztható elemeibõl áll L2.
nemoszt([], _K, []).
nemoszt([X|L1], K, L2) :-
        (   X mod K =:= 0 -> nemoszt(L1, K, L2)
        ;   L2 = [X|L3], nemoszt(L1, K, L3)
        ).

| ?- novo(1, 50, _L1), nemoszt(_L1, 2, _L2), nemoszt(_L2, 3, _L3),
nemoszt(_L3, 5, L).
L = [1,7,11,13,17,19,23,29,31,37,41,43,47,49] ? ;
No
| ?-
```



- Mihelyst `_L1 = [1|_]` lesz, indulhat a következő cél: `nemoszt(_L1,2_L2)`.
- A termelő és a fogyasztó (változó)függő ÉS kapcsolatban állnak. Egy hasonló struktúrájú program specializált végrehajtásában ésszerűnek látszik egy fifo buffer bevezetése a termelő és a fogyasztó rész közé, illetve indíthatunk párhuzamosan több fogyasztót, amennyiben az összes megoldást gyűjtjük, vagy bármelyik megfelel.
- lásd még Prolog jegyzet 174. oldal, egyszerűsített Hamming feladat

Keresési feladat

```
points_connected(A,B) :-
edge(A,X), points_connected(X,B).
```

Az edge által adott választási pontban adódó lehetőségeket párhuzamosan hajtjuk végre, ezt vagy-párhuzamosságnak nevezzük. Kicsit összetettebb példa az [1] queens kódja.

A párhuzamosságok típusairól részletesen a 2003-mas előadás anyagában írt Ludányi Zoltán.

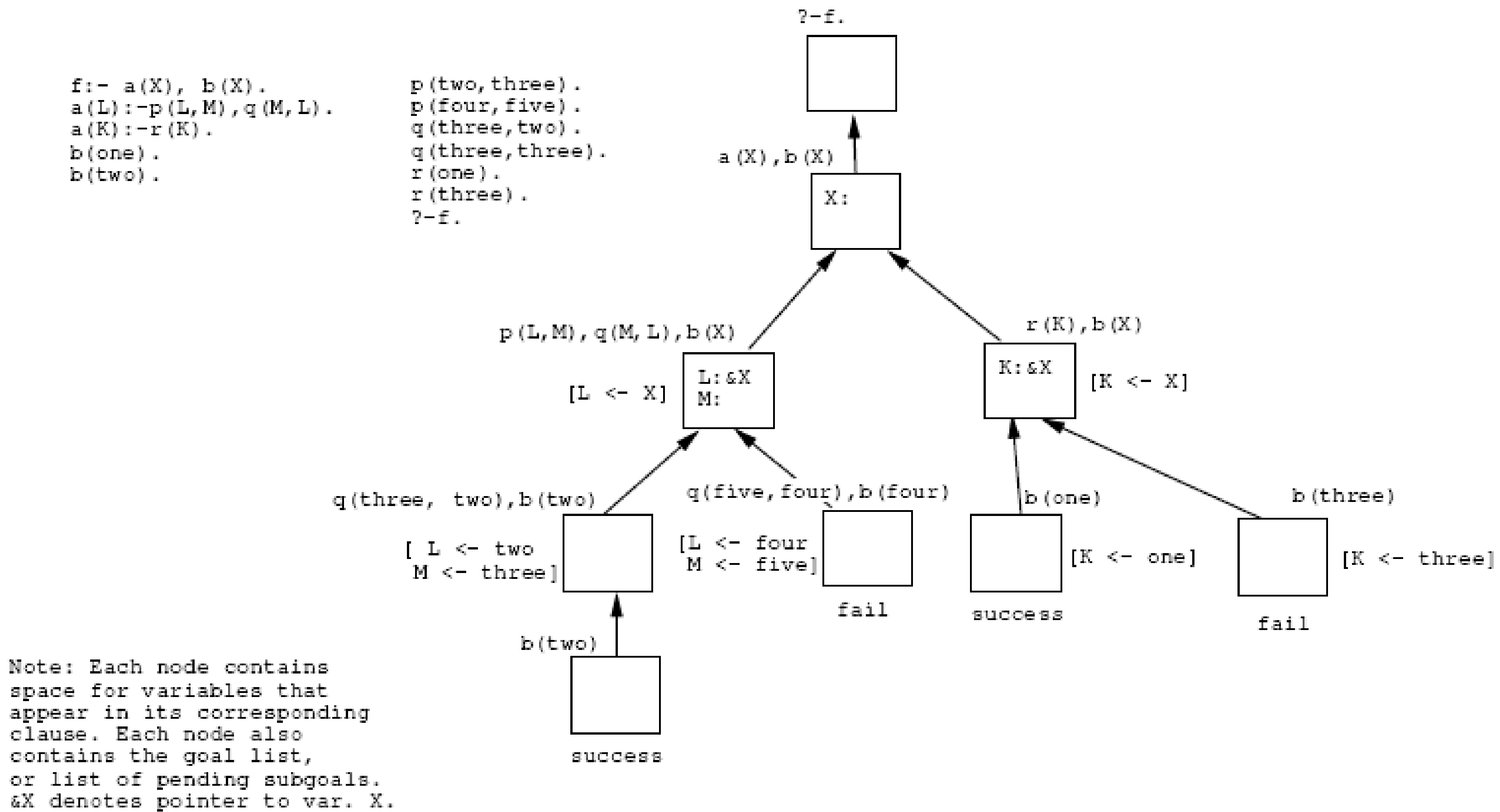


Figure 1: An Or-parallel Tree

Szorosan vagy lazán csatolt végrehajtás

- Lazán csatolt rendszerben nagyobb feladatokat éri meg szétosztani.
- Szorosan csatolt rendszer kevésbé skálázható, de már a személyi számítógépek ebbe az irányba fejlődnek.
- Egy ilyen rendszerben az operációs rendszerre támaszkodva, szálak vagy folyamatok létrehozásával használhatjuk ki a több processzort.
- A 'hétköznapi' grafikus eszközökben is szorosan csatolt párhuzamos végrehajtók jelennek meg (ps3), hatékony imperatív programozásukra jelenleg nincs bevált módszer.

Implicit/explicit párhuzamosítás

- Explicit esetben a programozó adja meg a párhuzamosítás helyét és módját, pl fork/join primitívekkel, vagy szálakkal, kritikus szakaszokkal.
- Implicit esetben a végrehajtó/interpreter dönt az elágaztatásról.

A logikai programozás nagy előnye, hogy forrásmódosítás nélküli (implicit) párhuzamos végrehajtása egyáltalán kivitelezhető. Imperatív nyelvek esetén ez igen izgalmas feladat. :)

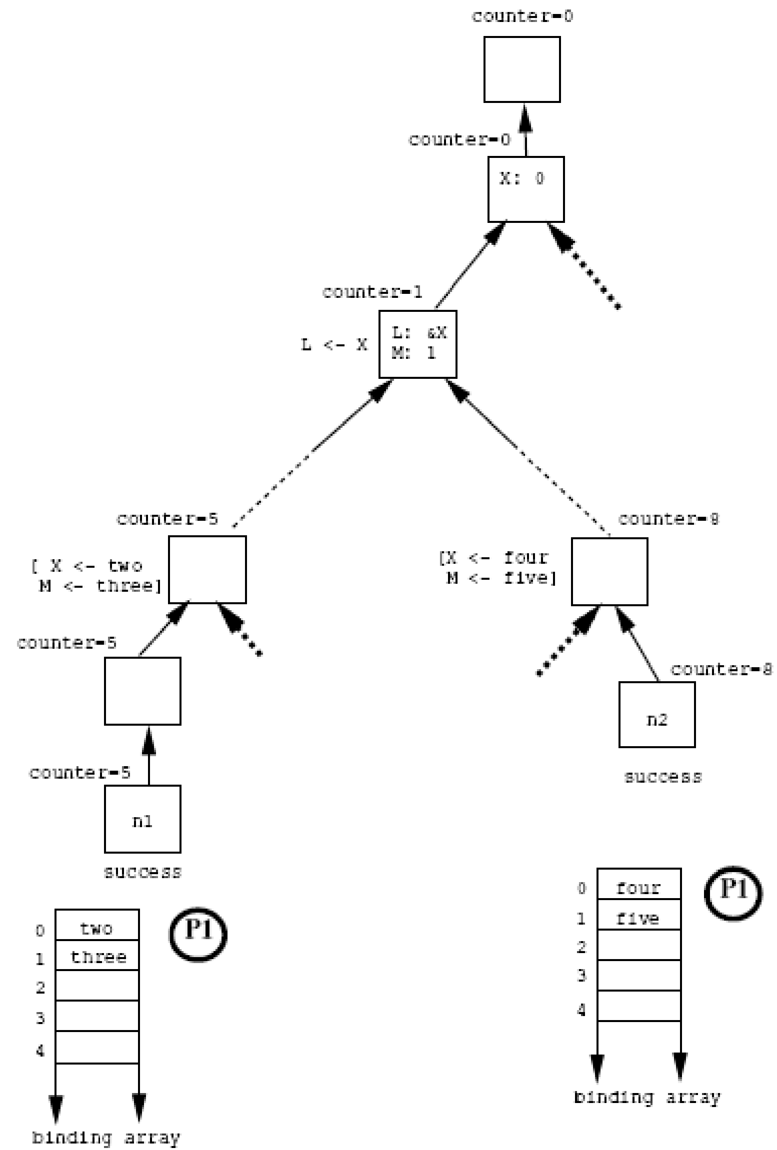
Vagy-párhuzamosság

Egy párhuzamos modellben több *végrehajtóegység* dolgozhat egy Prolog végrehajtás különböző részein. Egy nem dolgozó egységnek a *scheduler* (ütemező) keres új munkát a keresési fában.
A megvalósítás egyik fő nehézségét az egy adott keresési ághoz tartozó, *feltételes változókötések* (conditional binding) kezelése jelenti. Egy keresési ágon dolgozó végrehajtónak egy saját környezettel kell rendelkeznie, amelyben az ő ágához tartozó változókötéseket látja. Ennek számos megoldása létezik, alapvetően három kritériumot támasztunk ezekkel szemben:

1. Egy új környezet létrehozása legyen konstans idejű.
2. Változó hozzáférése és kötése legyen konstans idejű.
3. Egy feldolgozóegység feladatváltása legyen konstans idejű. (..)

Ezekből egyszerre csak kettő elégíthető ki. Az Aurora rendszerben implementált *Binding Arrays* módszerrel a feladatváltás ideje a két környezetben eltérő változókötések számától fog lineárisan függeni. Minden feldolgozóegységhez egy binding array tartozik. A feltételes kötésű változók sorszámot kapnak, és kötéseik a kötési tömb megfelelő helyére kerülnek.

A MUSE rendszer megvalósításában minden változókötés a feldolgozóegység saját vermében van, feladatváltáskor a felvett feladat vermének szükséges részét átmásolja a sajátjába.



counter=0
counter=0
X: 0
counter=1
L <- X
L: &X
M: 1
counter=5
[X <- two
M <- three]
counter=8
[X <- four
M <- five]
counter=5
counter=8
n2
success
counter=5
n1
success
0 two
1 three
2
3
4
P1
binding array
0 four
1 five
2
3
4
P1
binding array

A teljes Prolog nyelv támogatása

- A Prolog szekvenciális végrehajtásának „látszatát” szeretnénk megtartani
- Tiszta logikai programoknál ez nem okoz gondot.
- Problémát jelentenek a vágók és az egyéb mellékhatásos hívások.
- Külön figyelmet kell szentelnünk az ún. spekulatív hívások ütemezésére:

```
p(X, Y) :- q(X), !, r(Y).
p(X, Y) :- g(X), h(Y).
```

Az Aurora rendszerről nagyon röviden

- Interface az engine és az ütemező között
- Több ütemező
- A keresési fa privát és publikus részei
- Mellékhatásos predikátumok kezelése (várakoztatás)
- Mellékhatásos predikátumok nem-szinkron változatai (asynchronous és cavalier)

Az ütemező feladata a feldolgozókhoz munkákat rendelni, lehetőleg minimalizálva a környezetváltás költségét (távolságot a fában), növelve a granularitást (ne aprózza a munkát), illetve lehetőleg kerülve a spekulatív munkát.

A párhuzamos végrehajtás költségei

- A párhuzamos végrehajtás lehetőségének megteremtése a szekvenciális végrehajtásban is többletmunkát jelent.
- Az egyik konstans-idő kritérium megsértése többletmunkát jelent az adott helyen.
- Az Aurora és MUSE rendszerekben a feladatváltás költséges, a feladatváltásokat az ütemező kezeli, ezért a párhuzamos végrehajtásból adódó többletmunka igen nagy részéért az ütemező felelős, így az ütemező algoritmus kulcsfontosságú.

Ütemezési stratégiák

Argonne ütemező

- Minden node-ban helyi információt tartalmaz (van-e alatta munka) a munkások eligazítására.
- *Topmost dispatching*
- Minden csomópontban egy bitmap tárolja hogy alatta melyik munkás dolgozik, ez fontos vágás esetén.

Manchester ütemező

- Egy munkáshoz a hozzá legközelebb eső munkát próbálja rendelni, a távolság mértéke az átírandó kötések száma
- Ez ismét topmost dispatching-ot eredményez.
- Itt is worker-bitmap.

Wavefront ütemező

- Egy wavefront struktúra tárolja a legfelső csomópontokat.
- Egy munkás ebben talál következő munkát.
- Egy csomópont feldolgozása után a hullámfront kiegészül a következő párhuzamos csomópontra.

MUSE ütemező

- Bottom-most dispatching, minor task-switches.
- Ha munkát kérnek egy munkástól, a teljes private részét felszabadítja.
- A fa minden ágához egy *richness*t rendel, az ott található munka mennyiségét tükrözve.

Bristol ütemező [1]/4

- Másik érv a bottom-most dispatching mellett: a spekulatív munka kezelésének lehetősége.
- Először mindig nem-spekulatív munkát választ
- A spekulatív munkát "óvatosan kezeli", balról jobbra keres egy spekulatív fában, a spekulatív munkák között egyenletesen osztja szét a munkásokat.
- Egy spekulatív fában dolgozó munkás néha körbenéz, hogy van-e már nem spekulatív munka (*voluntary suspension*).
- Mellékhatásos predikátumok, cut és commit kezelése ...

Független és-párhuzamosság

```
fib(0,1).
fib(1,1).
fib(M,N) :- M1 is M - 1, fib(M1, N1),
            M2 is M - 2, fib(M2, N2),
            N is N1 + N2.
```

A függetlenség felismerése elég bonyolult feladat:

```
p(X,Y) :- r(X), s(Y).
?- p(Z,Z).
```

A probléma négy megközelítése

- In/out módok meghatároztatása, például:

```
p(+X,-Y) :- q(+X,-A), r(+X, -B), s(+A,+B,-Y).
```

- Worst-case fordítási idejű analízis
- Teljes futási idejű függőségnyilvántartás
- Hatékony középút teljes fordítási idejű és részleges futási idejű analízissel

Intelligens visszalépés:

```
?- a, b, c, d.
```

Ha b és c függetlenek és több megoldást adnak, c meghiúsulása esetén nem b-t, hanem a-t kell folytatnunk.

Teljes Prolog nyelv támogatása:

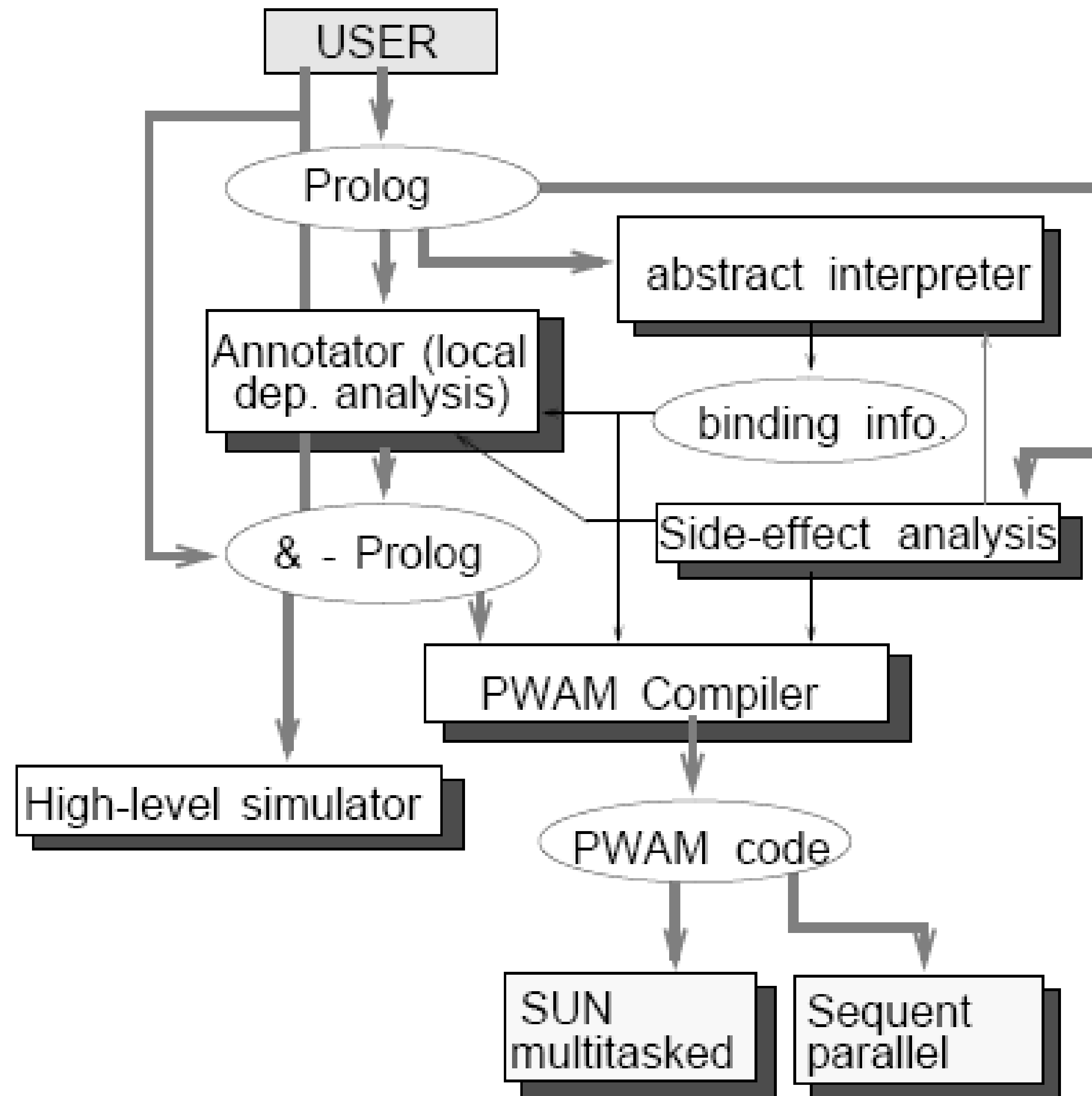
- A mellékhatásos predikátumok itt is gondot okoznak, pl a fenti példában ha b mellékhatásos.
- Spekulatív számítás: `q :- a, b.` számításában `a` meghiúsulása a `b` -be fektetett munkát feleslegessé teszi.

&-Prolog

Prolog kompatibilis, de nyelvi kiegészítéseket ad. Egy bemenő prolog kódot analizál, és, ahol tudja, párhuzamosítja, majd egy PWAM nevű architektúrán futtatja.

Egy egyszerű példa

```
p(X,Y) :-    q(X,Y), r(X), s(X).                      % Prolog
p(X,Y) :-    (ground(X) => q(X,Y) & r(X) & s(X) ).    % &-Prolog
p(X,Y) :-    (ground(X) -> q(X,Y) & r(X) & s(X)       % ‘=>’ kifejtése
             ; q(X,Y) , r(X) , s(X)
             ).
p(X,Y) :-    (ground(X) -> q(X,Y) & r(X) & s(X)       % Részletesebb
             ; q(X,Y), (ground(X) => r(X) & s(X))     %  vizsgálattal
             ).
```


USER
Prolog
abstract interpreter
Annotator (local dep. analysis)
binding info.
Side-effect analysis
& - Prolog
PWAM Compiler
High-level simulator
PWAM code
SUN multitasked
Sequent parallel

Példák az elemzésre és fordításra

```
matvecmul([Vect1|TVect],Vect2,[Res|TRes]) :-
           vecmul(Vect1,Vect2,Res),
           matvecmul(TVect,Vect2,TRes).

matvecmul([Vect1|TVect],Vect2,[Res|TRes]) :-
           ( ground([Vect2,Vect1,TVect]), indep(Res,TRes) =>
                  vecmul(Vect1,Vect2,Res) &
                  matvecmul(TVect,Vect2,TRes) ).
```

Ha az elemző bizonyítani tudja, hogy a predikátum hívásakor az első két argumentum mindig ground és a harmadik szabad változó, akkor elhagyja a (felesleges) feltételvizsgálatot:

```
matvecmul([Vect1|TVect],Vect2,[Res|TRes]) :-
               vecmul(Vect1,Vect2,Res) &
               matvecmul(TVect,Vect2,TRes).
```

A PWAM megvalósításról [3] ad összefoglalást.

A Parlog nyelv lehetőségeiről 2003-ban Ludányi Zoltán tartott előadást. (Vigyázat, itt a ',' és '&' jelentése fordított)

Függő és-párhuzamosság

- Változókötés függése
- Termelő és fogyasztó cél azonosítása
- Kétirányú kommunikációs csatorna

```
produce(O, Out) :- Out = [].
produce(N, Out) :-
        N > 0,
        N1 is N - 1,
        Out = [ferrari | Ms],
        produce(N1,Ms). consume([]).
consume([ferrari | Ms]) :-
        go_ride_ferrari,
        consume(Ms).

?-produce(10, X), consume(X).
```

A consume futása felfüggesztődik, amíg `Ms` nem kötött.

Kétirányú kommunikáció, coroutining

```
produce(0, Out) :- Out = [].
produce(N, Out) :-
        N > 0,
        N1 is N - 1,
        Out = [ferrari(Envelope) | Ms],
        spend(Envelope),
        produce(N1,Ms).

consume([]).
consume([ferrari(Envelope) | Ms]) :-
        Envelope = big_bucks,
        sell_ferrari_and_buy_beer,
        consume(Ms).

spend(big_bucks).
spend(small_bucks).

?-produce(10, X), consume(X).
```

Problémák a végrehajtáskor

• A fogyasztó nem köthet 'termelői' változót, fel kell függeszteni.

• Egyirányú kommunikáció esetén a helyzet a független esethez hasonló, csak a felfüggesztésekkel kell foglalkoznunk.

• Korutinos esetben problémát okoz a nemdeterminizmus. Három megoldást alkalmazhatunk:

  o Committed Choice (don't care): egy termelő és egy fogyasztó eredményt fogadunk el. A hívások “elkötelezik” (commit) magukat az egyik klózuk mellett kilépés előtt. (commit operátor ‘|’, hasonló a vágóhoz, de felfelé is vág)

  o Hagyjuk mindkét célt futni, amíg legfeljebb egy ágon sikeresek.

  o Minden kombináció kezelése.

Kombinált és-vagy párhuzamosság

Basic Andorra Model (Andorra-I)

- Az Aurora ütemezői használhatóak
- Nemdeterminisztikus cél nem indul el, amíg van determinisztikus, ezek párhuzamosan futnak. Lásd: Prolog jegyzet 192. oldal, 8.2.
- Ha már csak nemdeterminisztikus cél van, a legbaloldalibbra vagy-elágaztatás történik.
- Processzorcsoportok

Extended Andorra Model és AKL

- Az AKL nyelv teljes explicit párhuzamosításra ad lehetőséget.
- Nemdeterminisztikus cél is futhat, amíg konzisztens marad a külső környezettel, egyébként felfüggesztődik.
- Ha már csak felfüggesztett cél van, minden termelt változókötéshez másolat jön létre a fogyasztókból. (*non-deterministic promotion*)

```
p(X, Y) :- X = 2, m(Y).
p(X, Y) :- X = 3, n(Y).
q(X, Y) :- X = 3, t(Y).
q(X, Y) :- X = 3, s(Y).
r(Y) :- Y = 5.

?- p(X, Y), q(X, Y), r(Y).
```

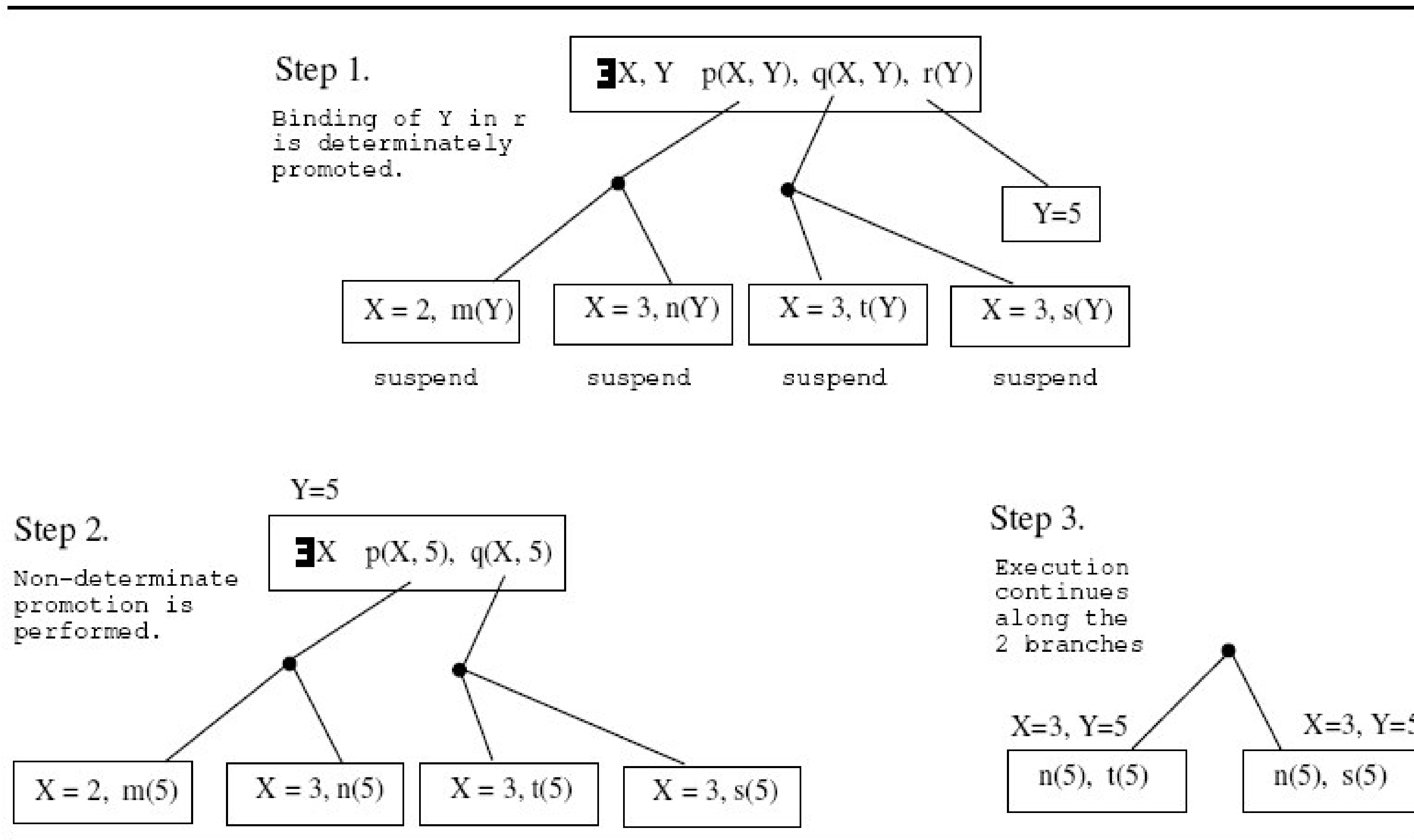


Figure 3: Execution in EAM

- Recompute - reuse dilemma, probléma a szekvenciális szemantikával
- PEPSys modell:
    - Hashing windows és timestamping
    - Független és-szakaszokat meg kell jelölni
    - Két és-párhuzamos célt futtat egyszerre, több esetén beágyaz
    - És ágon belüli vagy elágazás esetén keresztszoroz
    - A két ág megoldásai együttesen léteznek a memóriában, ezeket állnak párba újonnan érkező megoldásokkal, de bizonyos megoldások visszalépés áldozatai lehetnek, így néha újraszámolódnak, néha újrafelhasználódnak...
    - Nem is valósították meg :/
- AO-WAM: Binding arrays, base array, cross-product nodes ...

Különösen érdekesnek találtam . . .

- [1]/7: Exploiting Or-parallelism in Optimisation Problems

Irodalom:

[1] - Parallel Execution of Prolog Programs: A Survey. Gopal Gupta, Khayri A. M. Ali and Mats Carlsson, Manuel V. Hermenegildo

[2] - Various articles in: Contributions To Or-Parallel Logic Programming, Péter Szeredi 1997

[3] - The &-Prolog system: Exploiting Independent And-Parallelism, M. V. Hermenegildo, K. J. Greene