# Tabellázás és absztrakt kiértékelés XSB-vel

Szabó Péter
<pts+lakat@.bme.hu>

Válogatott fejezetek a logikai programozásból
kiseolőadás

2003. október




# Tabellázás

Ha egy algoritmus futása során egy (mellékhatás nélküli) függvényt többször hív azonos paraméterrel, akkor érdemes lehet a függvény visszatérési értékét megjegyezni (be-cache-elni), így az ismételt hívások gyorsabbak lesznek, mert számítás helyett a memóriából veszik elő a megjegyzett értéket. Ezt nevezzük *tabellázás*-nak (*tabular evaluation*).

A tabellázás úgy gyorsít, hogy a programkódot nem kell átírni, átszervezni miatta. Csak azt kell megadni, hogy mely függvényeket tabellázunk. Logikai programozásban függvények helyett eljárások szerepelnek, melyeknek mind bemeneti, mind kimeneti argumentumai tabellázásra kerülnek, akár behelyettesítettek, akár nem.




## Prolog példa ismétlődő függvényhívásra

```
fib(N, X) :- % ?X az +N-edik Fibonacci-szám
  ( N < 2 -> X = N
  ; N1 is N-1, N2 is N-2, fib(N1, X1), fib(N2, X2),
    X is X1+X2
  ).
```

Ez $f(N-2)$ értékét kétszer számolja ki, és a kisebb $f(I)$-ket még többször: $f(0)$ éppen $f(N)$-szer hívódik meg, tehát a futásidő exponenciális.

# Tabellázás példa

XSB (http://xsb.sourceforge.net) nyelven van tabellázás. Az alábbi példában csak az első sor változott az előző fib/2-höz képest.

```
:- table fib_tab/2. % új sor
fib_tab(N, X) :- % ?X az +N-edik Fibonacci-szám
  ( N < 2 -> X = N
  ; N1 is N-1, N2 is N-2, fib_tab(N1, X1), fib_tab(N2, X2),
    X is X1+X2
  ).

| ?- fib(33, X). % 5s fölött (800 MHz-es AMD processzoron)
X = 3524578 ;

| ?- fib_tab(33, X). % azonnal
X = 3524578 ;
```

# ⚜ Dinamikus programozás ⚜

SICStus-ban nincs tabellázás, ezért gondolkodni kell fib/2 gyorsításán:

```
fib_nr(N, X) :- fib_din(N, X, _Y).
fib_nr(N, X, Y) :- % X az N-edik, Y az (N-1)-edik Fib.-szám
  ( N < 2 -> X = N, Y = 0  % f(-1)=0 lesz
  ; N1 is N-1, fib_nr(N1, Y, Z), X is Y+Z
  ).
```

fib_nr/2 nem ágazik ketté, mivel $f(N-1)$-et és $f(N-2)$-t egyszerre számolja.

Az ún. *dinamikus programozás* ötlete az, hogy építsünk fel egy $N$ elemű tömböt a Fibonacci-számokból növekvő sorrendben, és olvassuk ki a tömb utolsó elemét. Az alábbi implementáció csak a tömb utolsó 2 elemét tartja nyilván. Vegyük észre, hogy jobbrekurzív.

```
fib_din(K, V) :- fib_din(K, V, 1, 0).
fib_din(N, V, X, Y) :-
  ( 1 = N -> V = X
  ; N1 is N-1, Z is X+Y, fib_din(N1, V, Z, X)).
```



# Összehasonlítás

A fenti példák azt érzékeltetik, hogy egy ügyesebb, de több programozói tudást és időt igénylő implementáció sokkal gyorsabb a naívnál. De majdnem ugyanekkora gyorsulás érhető el, ha a naív implementációt tabellázva futtatjuk. Persze a tabellázás jelentős memóriahasználattal jár, így pl. nem érdemes a member/2-t tabellázni.




# Többszörös visszatérés

A tabellázás megszünteti a többszörös visszatérést:

```
szuloje(peter, xy).
szuloje(peter, xx).
szuloje(pal, xy).
szuloje(pal, xx).

testvere(A, B) :- szuloje(A, C), szuloje(B, C).

| ?- testvere(peter, X).
X = peter;
X = pal;
X = peter;
X = pal; no
```

`:- table testvere/2.` esetén csak az első 2 választ kapnánk.

# Nincs végtelen ciklus

A tabellázás nem esik végtelen ciklusba, ha a függvény értékkészlete véges, de ha végtelen, akkor betelhet a memória.

```
:- table p/0, p/1.
p :- p.
p(X) :- X1 is X+1, p(X1).

| ?- p.
no % tabellázás nélkül végtelen ciklus lenne
| ?- p(0). % betelik a memória
```



# Útkeresés

Az irányított gráfban utat kereső algoritmust naivan is megvalósíthatjuk, nem lesz végtelen ciklus.

```
el(a, b).  el(b, c).  el(c, a).  el(c, d).

:- table ut/2.
ut(A, A).
ut(A, C) :- el(A, B), ut(B, C).

| ?- ut(d,a).
no
| ?- ut(a,d). % tabellázás nélkül végtelen ciklus
yes
```

# Útkeresés hosszal

Mivel a kimenő argumentumot ($N$) is tabellázzuk, ezért ut/3 az összes, kört nem tartalmazó utat megtalálja. Sajnos seta/3 értékkészlete végtelen, ezért végtelen ciklusba kerülhet.

```
em(a, b).  em(b, c).  em(c, a).  em(a, c).

:- table ut/3, seta/3.
ut(A, C, N) :- % A-ból C-be van N hosszú út
  ( C = A -> N = 0
  ; em(A, B), ut(B, C, N1), N is N1+1
  ).
seta(A, A, 0).
seta(A, C, N) :- % A-ból C-be van séta (esetleg körökkel)
  em(A, B), seta(B, C, N1), N is N1+1.

| ?- ut(a, b, N).
N = 1;
N = 2; no
| ?- seta(a, b, N). % végetelen ciklus
```



# Számlecsípő játék

Egy kicsit bonyolultabb feladat: adott számoknak egy listája. *Első* és *Második* felváltva vehet el a lista elejéről vagy végéről egy számot. Az nyer, aki nagyobb összeget szerez meg.

A megoldásra használt $O(2^n)$ idejű, rekurzív algoritmus: Nevezzük egy adott lista *különbség*-ének a nyeremény(*Első*) – nyeremény(*Második*) értéket, ha mindketten optimálisan játszanak. Az üres lista különbsége 0. Mivel *Második* optimálisan játszik, ezért nemüres lista esetén *Első* két különböző értékből választhat: balfej – különbség(balfarok) és jobbfej – különbség(jobbfarok). Ezek küzül *Első* a nagyobbat fogja választani.




# Egy rekurzív megoldás

```
% jat_ntl(+H, +AL, ?Dif): Ha a feladatot az AL lista első
% H eleme tartalmazza, akor a különbség Dif.
kulonbseg(0, _AL, Dif) :-
  !, Dif = 0.
kulonbseg(H, AL, Dif) :-
  AL = [A|As],
  nth(H, AL, B),
  H1 is H-1,
  kulonbseg(H1, As, A1), A2 is A-A1,
  kulonbseg(H1, AL, B1), B2 is B-B1,
  max(A2, B2, Dif).

| ?- kulonbseg(12, [1,2,3,4,5,6,7,8,9,10,11,12], Dif).
Dif = 6 ? ; no
```

kulonbseg/3 felgyorsítható tabellázással, mivel rengetegszer hívja önmagát, de $L$ csak $O(n^2)$ különböző értéket vehet fel.

## Teszteredmények

21 elemű listára a tesztgépen 6.5s-ig futott tabellázás nélkül, tabellázással mérhetetlenül gyorsan. Tabellázással 200 elemig fel lehetett menni, de 500 elemnél már elfogyott az 512Mb memória (és a 256Mb swap is).

Érdekes megemlíteni, hogy ha 16 bitre korlátozzuk a számok értékét, és 10240-re a lista hosszát, akkor a tabellázás memóriaigénye $4 \cdot 10240 \cdot 10240$ byte, azaz 400Mb, viszont az XSB már 500-as listahossznál is betelíti a memóriát, tehát nagyon pazarol. Az is érdekes, hogy az XSB tabellázás nélkül kb. 11-szer gyorsabban teljesített, mint a SICStus.




# A LAKAT véges modellje

Próbáljuk a tabellázást imperatív nyelven írt program optimalizására felhasználni. Ezt teszi a LAKAT.

Az egyszerű imperatív futtatási környezet jellemzői:

- véges sok változtatható változó van
- verem és memóriafoglalás nincs
- függvényhívás nincs, csak `goto` (és címkék)
- minden változónak van egy dinamikus (futásidejű) típusa
- az egyes típusok értékkészlete végtelen, de mi véges sok osztályra bontjuk őket (ettől lesz *abstract evaluation*), lásd később
- a program nemdeterminisztikus, mivel adatot olvashat be a felhasználótól

A fenti jellemzők biztosítják, hogy a program állapottere véges.




# Elemzési szempontok

Adott a program forráskódja. Arra szeretnénk választ kapni, hogy a program futásának befejeztével...

- kapott-e értéket a $V$ változó?
- felvehet-e a $V$ változó pozitív értéket? ... negatív értéket? ... lehet-e nulla?
- lehet-e a $V$ változó páros? ... páratlan?
- a program ráfutott-e valaha a $C$ : címkéjű utasításra?

Az optimalizálás szempontjából fölösleges tudnunk, hogy a program kerülhet-e végtelen ciklusba. Ezért a végtelen ciklust a meghiúsulással (azaz végzetes kivétel dobásával) ekvivalensnek tekintjük.

# Absztrakt példa

A feladatunk írni egy elemzőt, melynek bemenete egy program forráskódja, kimenete pedig a változók (absztrakt) értéke a végállapotban, továbbá a futás során elért címkék listája. Ha több megoldás lehetséges, akkor összeset ki kell írni.

Erre a programra:

```
write "Adj 1 egészt: ";
input n;
if 0 < n then goto pozitiv;
negalas -> let n = -n;
pozitiv ->
write "Abszolút értéke: ";
write n;
print "";
```

Ezt fogja kiírni:

```
ok, [negalas,pozitiv], [n=zero]
ok, [negalas,pozitiv], [n=p1]
ok, [        pozitiv], [n=p1]
ok, [negalas,pozitiv], [n=p2]
ok, [        pozitiv], [n=p2]
```

Ebből azt olvashatjuk ki, hogy a program mindig sikerül, a `pozitiv` címkére mindig rákerül a vezérlés, a `negalas` cimkére néha rákerül, néha nem, és az $n$ változó kimeneti értéke zero, p1, vagy p2.

## Az absztrakt kimeneti értékek jelentése

- *undefa*: a változó még nem kapott értéket
- *string*: a változó tetszőleges string
- *zero*: a változó 0 értékű
- *p1*: a változó pozitív páratlan szám
- *p2*: a változó pozitív páros szám
- *n1*: a változó negatív páratlan szám
- *n2*: a változó negatív páros szám

Tehát `ok`, `[pozitiv]`, `[n=p1]` azt jelenti, hogy $n$ a program futása után pozitív páratlan szám, és a program a `pozitiv` címkére ráfutott, de a `negalas` címkére nem.

## A LAKAT most

A véges állapottér miatt az absztrakt kiértékelés mindig véges időn belül sikerül. Az elemző kimenetét továbbadhatjuk a fordítóprogramnak, amely a binárisból kihagyhatja az elérhetetlen címkéket és az értéket soha nem kapó változókat.

A LAKAT project keretében eddig az elemző valósult meg XSB nyelven, tabellázás felhasználásával. A dokumentáció és az implementáció letölthető a `http://www.inf.bme.hu/~pts/lakat-latest.tar.gz` címről. Tervezem az egész újraírását, immár egy optimalizáló bytecode fordító formájában.




# A LAKAT használata

Egy `.lo` kiterjesztésű file-ba bele kell írni a programkódot, például:

```
% hetet.lo
ujra -> input i;
if i <> 7 then goto ujra;
```

Az XSB-t (`bin/xsb`-re végződő paranccsal) el kell indítani, majd `[lakat].` kérdéssel betölteni a `lakat.P`-t. Ezután:

```
lotest_abstract('hetet.lo'). % absztrakt kiértékelés
lotest_normal(  'hetet.lo'). % normál programfuttatás
```

A LAKAT helyesen kikövetkezti, hogy $i$ pozitív páratlan szám lesz kilépéskor.



# A fő predikátum

```
l0a_program(+Bind, +Prog, +ReachP, +Now, +Reach,
  -M, -Cind, -Seach).
```

Jelentése: a változók Bind értéke mellett, a Now címke által kijelölt pozíciótól kezdve futtatni a Prog programot, melyhez a címkék ReachP elérhetőségi listája tartozik, a címkék Reach elérési 0–1 listájából kiindulva; a program a változók értékét Cind-re módosítja, az M címkénél ér véget (M = stop/0 címke `stop` utasítás, M = stop/1 `throw` utasítás hatásársa keletkezik), és Seach lesz a címkék elérési listája (`stop/_`) nem szerepel benne).

Ez a predikátum van `:- table l0a_program/8.`-cal tabellázva, tehát az állapottér végessége miatt mindig lefut véges időben, és ugyanazt az eredmény nem adja ki többször. Óriási szerencse, hogy az XSB ilyen bonyolult, változókat is tartalmazó struktúrákkal meghívott predikátumokat is tud tabellázni. Jól látható a Now–M, Reach–Seach és Bind–Cind kimeneti–bemeneti változó megfeleltetés.

## ⁂ 1. korlát: túl bő választ ad INPUT esetén ⁂

Elvi korlát, hogy nem láthatjuk előre, mit gépel be a felhasználó, ezért az INPUT változója n2, n1, zero, p1 és p2 mindegyikét felveheti – pedig lehet, hogy a felhasználó mindig csak pozitív számot gépelne be stb.

## ⁂ 2. korlát: a többszörös siker lassít ⁂

```
input a;
let a = (a <= 42)*2+1;
```

Ebben a programban `a <= 42` értéke zero vagy p1, ezt p2-vel szorozva zero vagy p2-t kapunk, amihez p1-et hozzáadva mindenképpen p1 lesz az eredmény, de a predikátum kétszer vissza a p1 eredménnyel. Szerencsére a tabellázás elfedi a többszörös visszatérést.



## 3. korlát: túl bő válasz konstansok esetén

```
let a = 1;
let b = a + 1;
let c = (a < b);
```

A fenti utasítások hatására valódi módban $c$ értéke 1 lesz, absztrakt módban viszont zero és p1 is lehet, mert egy pozitív páratlan szám ($a$) lehet kisebb és nagyobb is egy pozitív páros számnál ($b$). Tehát az absztrakt mód egy hamis pozitív választ szült.

A problémát kiküszöbölhetnénk ún. *constant folding*-gal, azaz az utasítássorozatot (feltéve, ha nincs a közepén címke) már absztrakt kiértékelés előtt tarnszformálhatnánk így:

```
let a = 1;
let b = 2;
let c = 1;
```



# 4. korlát: túl bő válasz információvesztéskor

```
input a; % ebben különbözik az előző példától
let b = a + 1;
let c = (a < b);
```

A 2. sor következménye `a < b`, de ezt az információt a 2. sor végrehajtása során nem állítjuk elő, és nem rögzítjük, ezért a 3. sorban a és b is n2, n1, zero, p1 és p2 bármelyike lehet.

Megoldás: Kibővíthetnénk az absztrakt kiértékelés állapotterét változópárokra vonatkozó információval, de mivel a bővítés véges, és a valódi értékek halmaza végtelen, ezért mindig lenne információvesztés, és túl bő válasz.

Az egész számok 5 kategóriáját azért így választottam meg, hogy a logikai értékek (0 és 1) különböző kategóriába essenek, továbbá a fordító tudjon előjelre vonatkozó következtetést levonni, például: ha egy változót C-ben `signed int`-nek deklaráltak, de kikövetkeztettük, hogy sosem lesz negatív, akkor dolgozhatunk vele `unsigned`-ként. Persze C nyelvnél vigyázni kell, mert az modulo $2^N$ aritmetikában dolgozik, így pozitívak összege is lehet negatív.

## 5. korlát: csak a 0 őriz meg információt

```
ujra -> input i;
if i < 7 then goto ujra;
if 7 < i then goto ujra;
```

A fenti program sosem érhet véget úgy, hogy i páros (mivelhogy csak $i = 7$ esetben ér véget), a LAKAT mégis megengedi az $i = \mathrm{p2}$ hamis pozitív választ, mert túl szűk az absztrakció, és a 3. sor után nem tudjuk előállítani azt az információt, hogy most $i = 7$. 7 helyett 0-val viszont csak a zero válasz adódik.




# A nullára leálló program

```
ujra -> input i;            % 1. sor
if i < 0 then goto ujra;    % 2. sor
if 0 < i then goto ujra;    % 3. sor
```

Állapotátmeneti gráfja:

```
undef-1 -> n2-2 -> n2-1 -> n2-2...
        -> n1-2 -> n1-1 -> n1-2...
        -> p1-2 -> p1-3 -> p1-1 -> p1-2...
        -> p2-2 -> p2-3 -> p2-1 -> p2-2...
        -> zero-1 -> zero-2 -> zero-stop
```

Itt a gráf csúcsaiban az i változó absztrakt értéke és a végrehajtásra váró sor száma szerepel. A valódi állapotátmeneti gráf csúcsaiban még az eddig elért címkék listája is helyet kap.

A `...`-os ágak nem állnak le, az egyetlen leálló ág a `zero-stop`, tehát végül $i$ a zero osztályba esik.



## 6. korlát: áttekinthetlen, ömlesztett eredmény

Gyakran nehéz levonni hasznos, de nem triviális eredményt a hosszú adatsorból, amit a LAKAT kiad. Ha az összes ág sorait bagof/3-mal kigyűjtjük, és aggregáljuk, akkor elvész pl. az az információ, hogy az alábbi példában az $a$ és $b$ változók paritása megegyezik, tehát csak a [b=p1,a=p1], [b=p1,a=n1], [b=p2,a=p2], [b=p2,a=zero], [b=p2,a=n2] kötések fordulhatnak elő.

```
input a;
let b = a*a+2;
```

Ha a LAKAT-ot függvények optimalizására kívánjuk használni, akkor elvárható, hogy `h() { #1 #2 }`-t is ugyanolyan hatékonyan optimalizálja, mint `f() { #1 g(); } g() { #2 }`-t; de $f$ és $g$ közös változóiról (azaz $h$ változóiról) szerzett aggregált információ kevesebb, mint ha $f$-et és $g$-t külön vizsgálnánk.

# A készülő LAKAT-1 jellemzői

- a C nyelven definiált függvények egy igen szűk részhalmazát optimalizálja (nem csak elemez)
- C-ről C-re fordít
- belső bytecode reprezentációt használ
- automatikus constant folding, még az absztrakttá alakítás előtt
- törli az elérhetetlen kódrészleteket
- törli a fel nem használt változóka
- jelzi, ha egy változót értékadás nélkül használunk
- egyéb egyszerű típusok, pl. double is megengedettek
- állítható túlcsordulás-kezelő politika
- elemzés és törlés egymást váltogatja, amíg változik a kód
- az elemzés látható kimenetében csak a deklarált címkéket mutatja

Egyéb ötlet: intervallumok képzése a programkódban szereplő összes konstans alapján