Elosztott deklaratív programozás Oz nyelven

Szoboszlay Dániel

2005.11.07.

Mi is az az Oz nyelv?

- „Az Oz egy magas szintű programozási nyelv, amelyet a modern, fejlett konkurens, intelligens, elosztott, soft real-time, párhuzamos, interaktív és pro-aktív alkalmazások igényeihez fejlesztettek ki.”

Mi is az az Oz nyelv?

- „Az Oz egy magas szintű programozási nyelv, amelyet a modern, fejlett konkurens, intelligens, elosztott, soft real-time, párhuzamos, interaktív és pro-aktív alkalmazások igényeihez fejlesztettek ki.”
- Konkurens adatfolyam nyelv, amely nagy számú szál futtatására képes egy megosztott tár fölött.
- Automatikus szinkronizáció az adatok rendelkezésére állása szerint.
- A Mozart rendszer átlátszó módon támogatja hálózatba kötött gépek elosztott programozását.
- Számos programozási paradigma támogatása: objektum orientált, funkcionális és logikai programozásra is lehetőséget nyújt.

Az Oz kernel nyelve

<utasítás> ::= <utasítás> <utasítás>
| $X = f(l_1 : Y_1 \ldots l_n : Y_n)$
| $X =$ <szám, atom, boolean>
| {NewName X}
| $X = Y$
| local $X_1 \ldots X_n$ in S end
| proc {X $Y_1 \ldots Y_n$} S end
| {X $Y_1 \ldots Y_n$}
| {NewCell Y X}
| $Y =$ @X
| $X := Y$
| {Exchange X Y Z}
| if B then S_1 else S_2 end
| thread S end
| try S_1 catch X then S_2 end
| raise X end

Elsődleges típusok

- Value
  - Number
    - Int
    - Float
  - Record
    - Tuple
  - Procedure
  - Cell
  - Chunk
    - Class
    - Object
    - ⋮
    - Lock
    - Port
  - Space
  - Thread
  - ByteString
  - BitString
- FDInt
- Char
- Literal
  - Atom
  - Name
    - Bool
    - Unit

Elsődleges típusok

Számok

5 ~2 072 0xFC 3.14 ~2.6e~1 &t &\n &\071

Literálok

```
a foo 'Hello World' {NewName Y} true false unit
```

Rekordok

```
tree(key:apple value:2 left:leaf right:leaf)
bar(fun:proc {Fun I Ret} Ret=I+1 end)
my_tuple(4.5 X) 1#2 1#2#three
```

Chunks

```
P={Port.new S} {Port.send P 1}
```

Hasonlóak a rekordokhoz, de nem minden feature-jük érhető el: absztrakt adattípusok!

Másodlagos típusok

Feature-ök

```
key left 1 2
```

Egészek vagy literálok

Listák

```
nil 1|2|3|nil [1 2 3] "string" '|'(1 '|'(2 X))
```

A ciklikus listák is megengedettek!

Virtuális füzérek

```
"A vonat " # 18 # ':' # "00" # "-kor " # indul
```

Atomok, füzérek, bájtfüzérek és számok a # operátorral összekapcsolva.

Modulok

A modulok olyan rekordoknak tekinthetők, melyeknek a kívülről látható eljárások, objektumok és értékek a feature-jeik.

Egyszerű modul

```
declare List in
local
   proc {Append ...} ... end
   proc {MergeSort ...} ... end
   proc {Sort ...} ... {MergeSort ...} ... end
in
   List = 'export'(append:Append sort:Sort)
end
```

Funktorok

A modulok létrehozására a funktorok szolgálnak, ezek a Chunkok közé tartozó adatszerkezetek.

Modul létrehozása funktorral

```
functor
export
   append:Append
   sort:Sort
define
   proc {Append ...} ... end
   proc {MergeSort ...} ... end
   proc {Sort ...} ... {MergeSort ...} ... end
end
```

Osztályok és objektumok

- Egy osztály egy Chunk, annak a leírása, hogy az osztályba tartozó objektumok milyen attribútumokkal és metódusokkal bírjanak.
- Az objektumok unáris függvények, argumentumuk egy rekord, melynek címkéje a meghívandó metódus neve, mezői a metódus „argumentumai”.
- Az attribútumokból az adott objektum példány által elérhető cella rekord lesz, a rekord feature-jei az attribútumok nevei.
- A metódusok függvények 3 argumentummal:
  - A metódus „hívásakor” az objektum függvénynek átadott rekord.
  - Az objektum attribútumainak rekordja.
  - Az adott példány (azaz az objektumfüggvény).

Osztályok

Osztály leírása szintaktikai édesítőszerek nélkül

```
declare Counter
local
   Attrs = [val]
   MethodTable = m(init:Init inc:Inc)
   proc {Init M S Self}
      init(Value) = M in (S.val) := Value
   end
   proc {Inc M S Self}
      X inc(Value)=M in X = @(S.val) (S.val) := X+Value
   end
in
   Counter = {NewChunk c(methods:MethodTable attrs:Attrs)}
end
```

Osztályok

Osztály leírása `class` szerkezettel

```
class Counter
   attr val
   meth inc(Value)
      val := @val + Value
   end
   meth init(Value)
      val := Value
   end
end
```

A `class` szerkezet fejlettebb lehetőségei

A metódus definíciókon belül:

- az állapot rekord mezőinek elérése implicit;
- az objektum példány elérhető a `self` kulcsszóval;
- a metódusok az alábbi szerkezettel is meghívhatók: `Class , method(...)`

Egyéb programozási megoldások is támogatottak:

- (többszörös) öröklődés;
- feature-ök az objektumokban;
- megadhatóak a feature-ök, attribútumok és metódus argumentumok alapértelmezett értékei;
- paraméteres osztályok: osztállyal visszatérő eljárások;
- private és protected metódusok;
- kritikus szakaszok a metódusokban.

Objektumok

Objektumok létrehozása

```
proc {New Class InitialMethod ?Object}
   State O
in
   State = {MakeRecord s Class.attrs}
   {Record.forAll State proc {$ A} {NewCell _ A} end}
   proc {O M}
      {Class.methods.{Label M} M State O}
   end
   {O InitialMethod}
   Object = O
end
```

Példa objektumok és osztályok használatára

```
declare C in
C = {New Counter init(0)}
{C inc(2)}
local X in
   thread {C inc(X)} end
   X=5
end
```

Szálak

- Szálak indítása: `thread` S `end`
- A szálak használata nagyon olcsó, akár 100.000 szál konkurens futtatása sem ró jelentős többletmunkát a rendszerre.
- Soft real-time funkciók: `Time` modul:
  - {`Alarm` I U} I ms elteltével U-t egyesíti a `unit` értékkel.
  - {`Delay` I} legalább I ms-ra felfüggeszti az aktuális szál futását.
- Szinkronizációs eszközök:
  - Adatfolyam jelleg: a műveletek az általuk igényelt változók kötéséig blokkolnak;
  - `General` modul: a változók kötése és igényeltségük egyaránt lekérdezhető, ezen eseményekre szinkronizálni lehet;
  - `Exchange` művelet: cellák oszthatatlan írása és olvasása;
  - Jövők használata.

Szinkronizáció

Szinkronizáció blokkoló műveletekkel

```
proc {Producer Xs}
   case Xs of X|Xr then X=something {Producer Xr}
   []         nil  then {Browse 'producer out'}
   end
end
proc {Consumer N Xs}
   if N<=0 then Xs=nil {Browse 'consumer out'}
   else X|Xr = Xs in
      {Browse 'consumed '#X}
      {Consumer N-1 Xr}
   end
end
{Consumer 100 {Producer}}
```

Szinkronizáció a General modullal

```
proc {Worker W S}
   thread
      {Delay 10}
      S=W
   end
end
declare Sig in
   {Worker {Worker {Worker unit}} Sig}
   {Wait Sig}
```

Jövők

- Létrehozás: `Y = !!X`
- A jövő egy logikai változóhoz kötött, csak olvasható változó. Azaz az írási műveletek a jövőn blokkolódnak, amíg az eredeti változó értéket nem kap: ekkor olvasási műveletté válnak.
- Ha a jövő értéke igényelt lesz, az eredeti változó értéke is igényelt lesz.

Alapfogalmak

Hely (*site*): a Mozart rendszer egy futú példánya, egy OS process. A helyek közötti kommunikáció szempontjából lényegtelen, hogy egy gépen, vagy két, hálózatba kötött gépen futnak-e.

Jegy (*ticket*): egy Mozart erőforráshoz tartozó, globálisan egyedi szöveges azonosító. A jegyek segítségével érheti el egy site a másik erőforrásait.

Befőtt (*pickle*): állapotmentes adatokat tartalmazó fájl. A befőttek írhatóak és (akár URL-ekről is) beolvashatóak, így szintén kapcsolatot teremthetnek távoli helyek között.

Az elosztott alkalmazások több helyen futnak, és közös erőforrásokra hivatkoznak. Az alkalmazások szempontjából (néhány kivételtől eltekintve) átlátszó, hogy hány helyen futnak.

Szinkronizáció

- Az állapotmentes értékek egyszerűen mozgathatóak a helyek között.
- Az állapottal bíró értékek (pl.: cellák) konzisztenciáját a rendszer automatikusan biztosítja.
- A Mozart rendszeren kívüli erőforrások (pl.: fájl leírók) nem értelmezettek távoli helyeken, nem mozgathatóak.
- A szálak azon a helyen futnak, amin létrehozták őket.

A kód végrehajtásának helye

- A függvények kódja is érték, ami átkerül az azt használó helyre.
- De néha szükség van távoli függvényhívásokra is!
- Megoldás: a végrehajtás alatt álló szálak már nem utaznak: elindítunk egy szálat a szerver helyén, ami igény esetén végrehajtja az eljárásunkat.
- A cél egy olyan `MakeStat` eljárást készíteni, ami egy tetszőleges eljáráshoz előállít egy vele ekvivalens szemantikájú, de stacionárius eljárást.

Unáris függvények távoli végrehajtása

```
proc {MakeStat Proc StatProc}
   S P={NewPort S}
   N={NewName}
in
   % Client side:
   proc {StatProc M}
   R in
      {Send P M#R}
      if R==N then skip else raise R end end
   end
   % Server side:
   thread
     {ForAll S proc {$ M#R} thread
try {Proc M} R=N catch X then R=X end end end}
   end
end
```

A `MakeStat` működése

- `P` egy port: `{Send P X}` az `X` értéket a porthoz tartozó nyílt végű listába írja (`MakeStat`-ban ez a lista `S`).
- `StatProc` egy unáris függvény lesz, ami átkerül a kliens oldalra.
- A kommunikáció a kliens és a szerver között a porton át történik. A portra párokat írunk:
  - `M` az eljáráshívás argumentuma
  - `R` az eljárás végrehajtása során fellépett kivétel vagy egy máshol még nem szereplő érték lesz, ha nincs kivétel.
- A szerver a portra írt minden egyes párra egy külön szálon végrehajtja az eljárást, és beállítja `R` értékét.
- A kliens szinkronizál az eljárás kezdetén és végén a szerverrel.

Unáris függvények aszinkron, soros végrehajtása

```
proc {MakeSAS Proc StatProc}
   S P={NewPort S}
in
   % Client side:
   proc {StatProc M}
      {Send P M}
   end
   % Server side:
   thread
      {ForAll S proc {$ M} thread
try {Proc M} catch X then skip end end end}
   end
end
```

Modulok

`Connection:` jegyek kezelése.

`Pickle:` befőttek megvalósítása.

`Remote:` új helyek létrehozása.

`Fault:` hiba detektáló és -kezelő eszközöket biztosít.

Connection modul

`{Connection.offer` *X Ticket*`}`: *Ticket* egy jegy lesz, amin keresztül *X*-et egyszer el lehet kérni.

`{Connection.offerUnlimited` *X Ticket*`}`: *Ticket* egy jegy lesz, amin keresztül *X*-et akárhányszor el lehet kérni.

`{Connection.take` *Ticket X*`}`: *X* a *Ticket* jegyben felkínált érték lesz.

Jegyet csak nem helyhezkötött értékről lehet készíteni.

A jegyek szövegesen kódolható értékek. Pl.:
`'x-ozticket://134.96.186.115:9000:egbj0:DS/v:s:kn'`

Pickle modul

Alapvető funkciók:

`{Pickle.save` *X Path*`}`: *X*-et kiírja a *Path* elérési úton található fájlba.

`{Pickle.load` *Url X*`}`: *X*-et beolvassa az *Url* URL-lel azonosított fájlból.

Befőttbe cask állapotmentes értéket lehet írni!

Néhány általánosabb funkció:

`{Pickle.saveCompressed` *X Path Level*`}`: *Level* egy 0 és 9 közötti szám lehet, ami a tömörítés mértékét írja elő.

`{Pickle.saveWithHeader` *X Path Header Level*`}`: a befőtt elejére írja a *Header* virtuális füzérben adott fejlécet. (Ez lehet pl. egy shell script a befőtt betöltésére, vagy egy komment.)

`{Pickle.loadWithHeader` *Url P*`}`: *P* egy pár lesz, melynek első eleme a fejléc, a második pedig a beolvasott érték.

`{Pickle.pack` *X S*`}`: *X*-et az *S* bájtfüzérbe írja ki.

`{Pickle.unpack` *S X*`}`: *X*-et beolvassa az *S* bájtfüzérből.

Remote modul

A `Remote` csomag a `Remote.manager` osztályt tartalmazza. Az osztály minden példánya egy létrehozott helyet jelent.

Néhány fontos metódus:

`init(host:` *H* `fork:` *F* `detach:` *D* `)` új hely létrehozása a *H* gépen.

- A létrehozás az `ozengine` program elindítását jelenti.
- Az indításhoz az *F* parancsot használja a rendszer, pl.: `'rsh' 'ssh'`.
- Ha *D* `false`, a létrehozott folyamat leáll, ha a szülője leállt, egyébként tovább működik.

`ping()` ha a hely nem válaszol, kivételt vált ki.

`close()` a hely tervezett leállítása.

Helyek használata

Egy `init`-tel létrehozott `Remote.manager` példány még nem csinál semmit. Új feladatot az `apply` metódussal adhatunk neki, egy funktor segítségével:

`apply(` *Functor Modul* `)`

Functor a funktor, ami a végrehajtadnó műveletet tartalmazza.

Modul a visszatérési érték, a létrehozott modul.

Egy egyszerű példa

```
declare R F M
R={New Remote.manager init(host:"192.168.0.5")}
F=functor export x:X define X={Fact 30} end
M={R apply(F $)}
{Browse M.x}
```

Az előbbi példa

```
declare R F M
R={New Remote.manager init(host:"192.168.0.5")}
F=functor export x:X define X={Fact 30} end
M={R apply(F $)}
{Browse M.x}
```

Funktor külső referenciával

```
declare R F M X
R={New Remote.manager init(host:"192.168.0.5")}
F=functor define {Fact 30 X} end
M={R apply(F $)}
{Browse X}
```

Az Oz deklaratív lehetőségei

- Az Oz lehetőséget biztosít a deklaratív programozásra, mert vannak logikai változói. Ezek működése hasonló a Prolog változóéihoz.
- Azonban az Oz adatfolyam nyelv, nincs benne visszalépéses keresés!
- Az Oz a keresés megvalósítására a számítási terek fogalmát vezeti be.
- A számítási terek révén a probléma megfogalmazása és a keresési technika szeparálható.

A logikai változók kényszer szemantikája

- A logikai változókról tárolt információ: $X_1 = U_1 \ldots X_n = U_n$, ahol

  X_i : egy logikai változók;
  U_i : vagy egy logikai változó vagy egy Oz entitás.

- Ez az információ egy logikai formulának is tekinthető: $\exists Y_1 \ldots Y_m : X_1 = U_1 \wedge \ldots \wedge X_n = U_n$
- A logikai változókra felírt kényszerek mellett keressük a formula egy megoldását: CLP feladat. Az egyszerű kényszereket az egyesítések, az összetetteket a szálak valósítják meg.
- Az Oz futása közben így háromféle tárat használ:

  Kényszer tár : logikai változók és kötéseik.
  Cella tár : az állapottal bíró cellák, objektum állapotok tára.
  Eljárás tár : a változókat használó eljárások kódja.

Számítási terek

- A tárak és a futó szálak együttese a számítási tér.
- A számítási terek egy fát alkotnak: egy szál létrehozhat altereket, melyek szülője a szál tere lesz.
- Minden szál és változó pontosan egy számítási térhez tartozik. De a leszármazott terekből látszanak (olvashatóak) a felmenőik változói.
- Egy számítási térben azon változókra lehet kényszereket megadni, amelyek abból látszanak. Ezek a kényszerek viszont csak ebben a térben és a leszármazottaiban lesznek láthatóak.
- Egy számítási tér meghiúsult, ha kényszer tára logikai ellentmondásban van a szülője kényszer tárával.
- Egy számítási tér összefésülhető a szülőjével: ekkor az altér megszűnik és minden tartalma átkerül a szülőbe.

A számítási terek állapotai

- Egy tér futtatható, amíg maga vagy egy altere tartalmaz egy futtatható szálat, egyébként blokkolt.
- Egy blokkolt tér stabil, ha a felmenő tereiben semmilyen új kényszer nem tehetné ismét futtathatóvá. Ilyenkor a tér lehetséges alállapotai:

  Megoldott: a térben nincsen választási pont.

  Elágazó: a térben fut (1 db) szál, ami egy választási ponton vár legalább 2 alternatívával.

  Meghiúsult: a tér által képviselt alternatíva ellentmondásra vezetett.

  Összefésült: az összefésült tér elérte életciklusa végét.
- Egy blokkolt, de nem stabil tér felfüggesztett állapotban van. Ez tipikusan csak a konkurens működés miatt előállt átmeneti állapot.

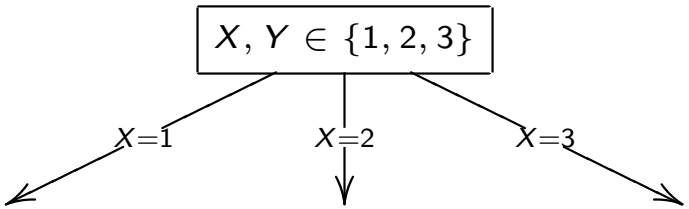
Keresési és disztribúciós stratégiák

Egy keresési probléma megoldásának általános menete:

- A kényszereket leíró kötésekkel és szálakkal létrehozunk egy számítási teret.
- A szálak erősítik a kényszer tárat, amíg tudják.
- A probléma jellegétől függő módon valamelyik szál létrehoz egy elágazási pontot néhány alternatívával, és felfüggesztődik. Ez a disztribúciós stratégia.
- A keresést végző függvény a számítási téren kívül észleli, mikor a tér elágazó állapotba ér. Ekkor minden egyes alternatívára létrehozhatja a tér egy másolatát, és azokban az egyes ágakon folytatódhat a végrehajtás.
- Az, hogy a választási pontokat és azok egyes alternatíváit milyen sorrendben fejtjük ki, független a megoldandó problémától! Ez a keresési stratégia.

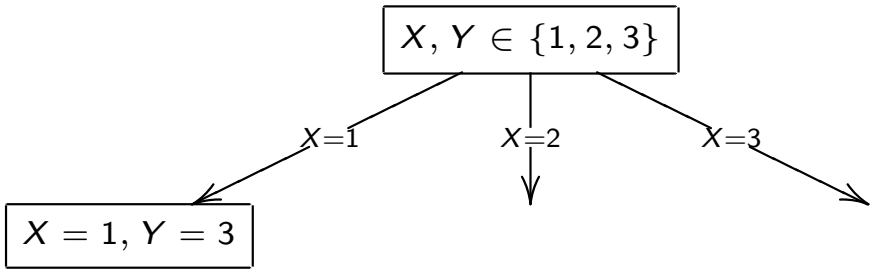
Példa: $X, Y \in \{1, 2, 3\}$ $X + Y = 4$

Felsorolásos disztribúciós stratégia



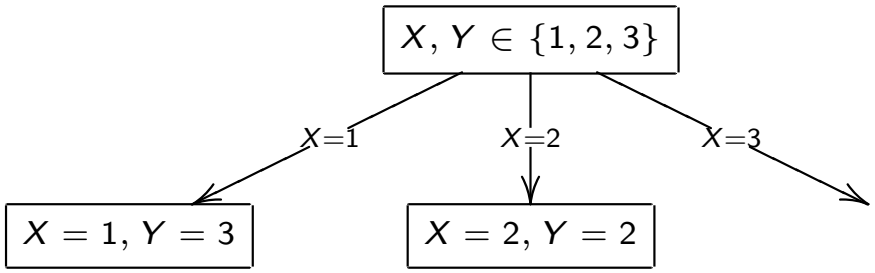
Példa: $X, Y \in \{1, 2, 3\}$ $X + Y = 4$

Felsorolásos disztribúciós stratégia



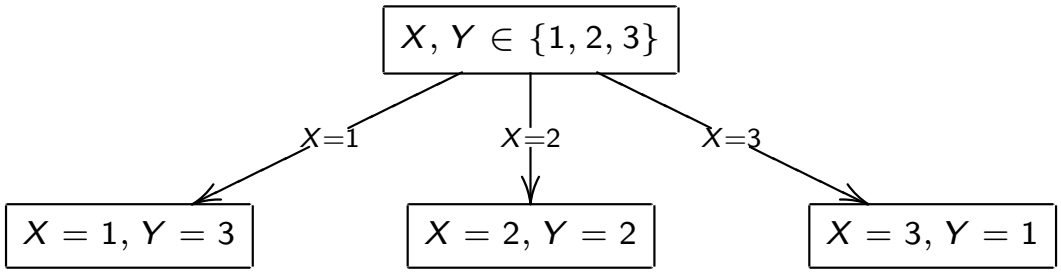
Példa: $X, Y \in \{1, 2, 3\}$ $X + Y = 4$

Felsorolásos disztribúciós stratégia



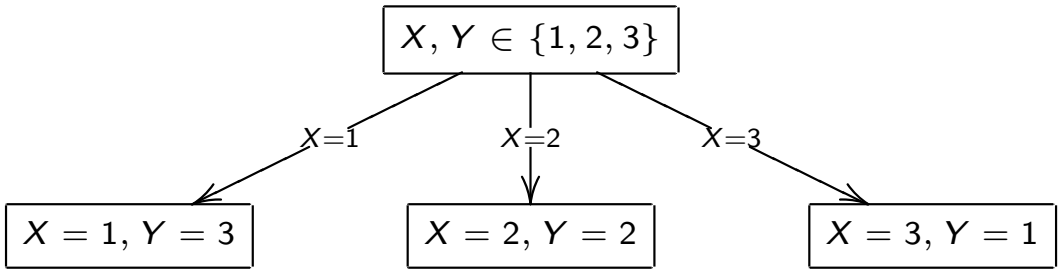
Példa: $X, Y \in \{1, 2, 3\}$ $X + Y = 4$

Felsorolásos disztribúciós stratégia

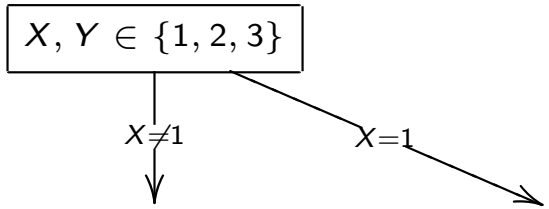


Példa: $X, Y \in \{1, 2, 3\}$ $X + Y = 4$

Felsorolásos disztribúciós stratégia

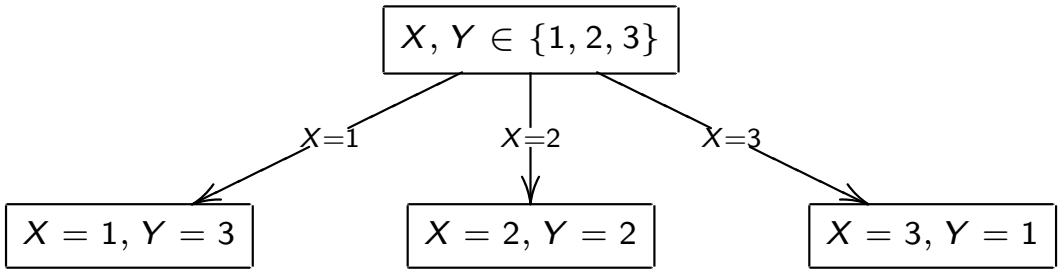


Lépegető disztribúciós stratégia, mélységi kereséssel

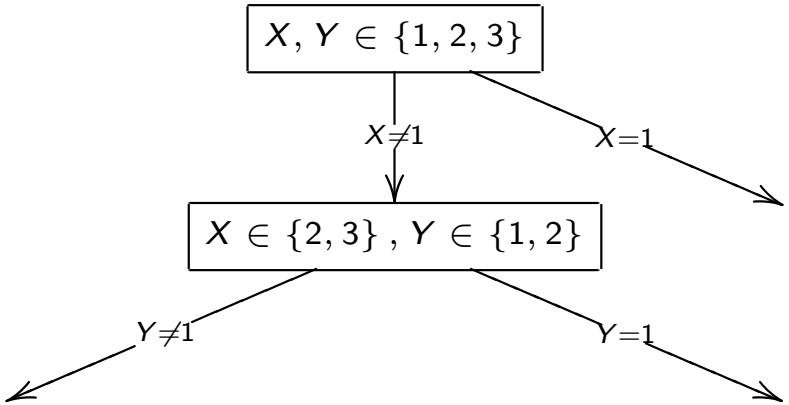


Példa: $X, Y \in \{1, 2, 3\}$ $X + Y = 4$

Felsorolásos disztribúciós stratégia

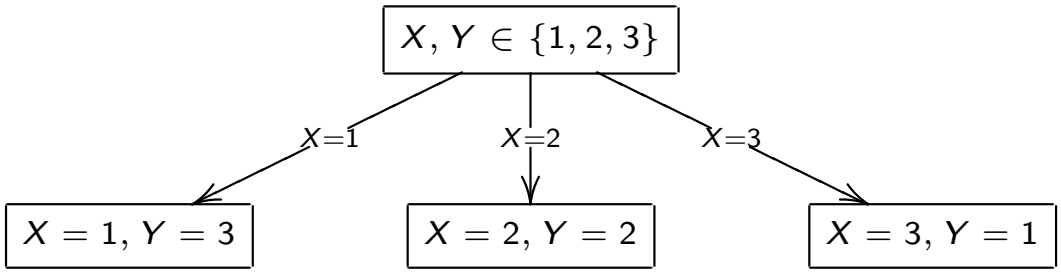


Lépegető disztribúciós stratégia, mélységi kereséssel

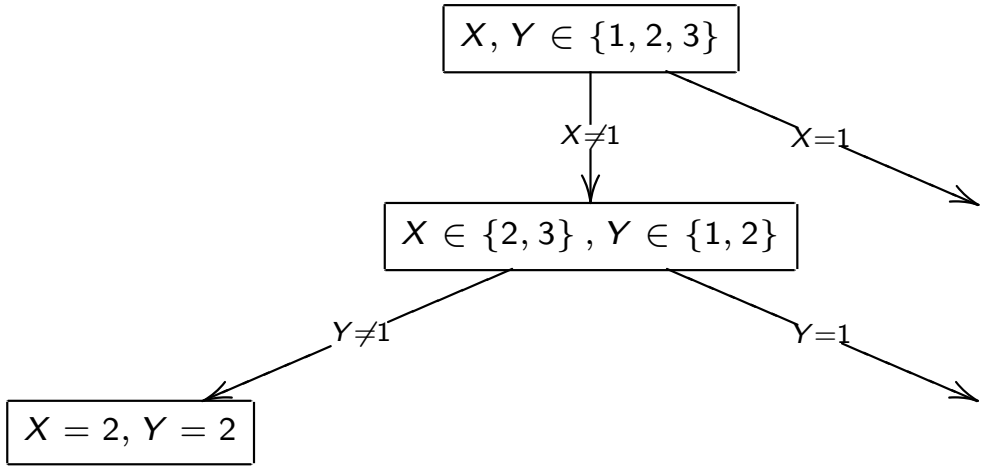


Példa: $X, Y \in \{1, 2, 3\}$ $X + Y = 4$

Felsorolásos disztribúciós stratégia

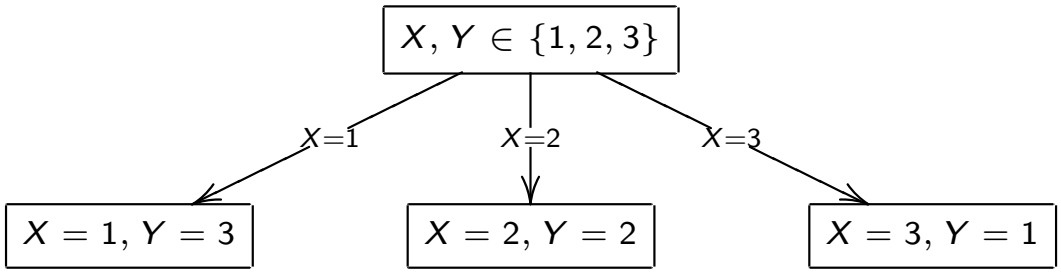


Lépegető disztribúciós stratégia, mélységi kereséssel

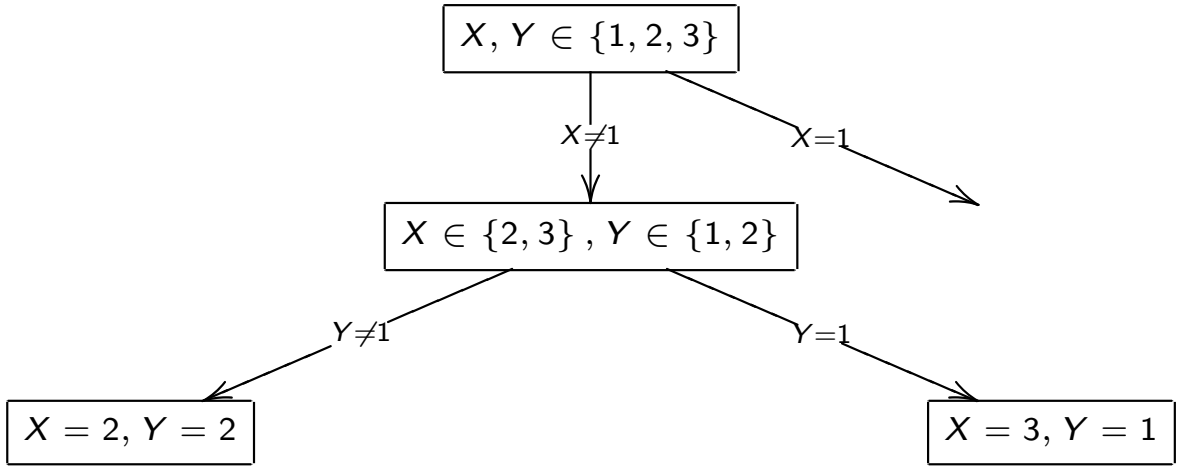


Példa: $X, Y \in \{1, 2, 3\}$ $X + Y = 4$

Felsorolásos disztribúciós stratégia

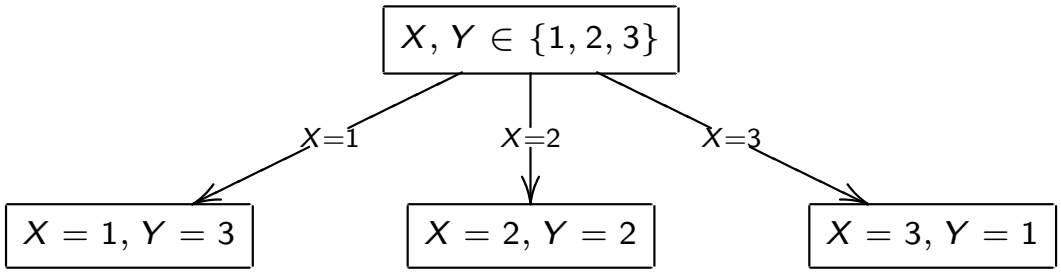


Lépegető disztribúciós stratégia, mélységi kereséssel

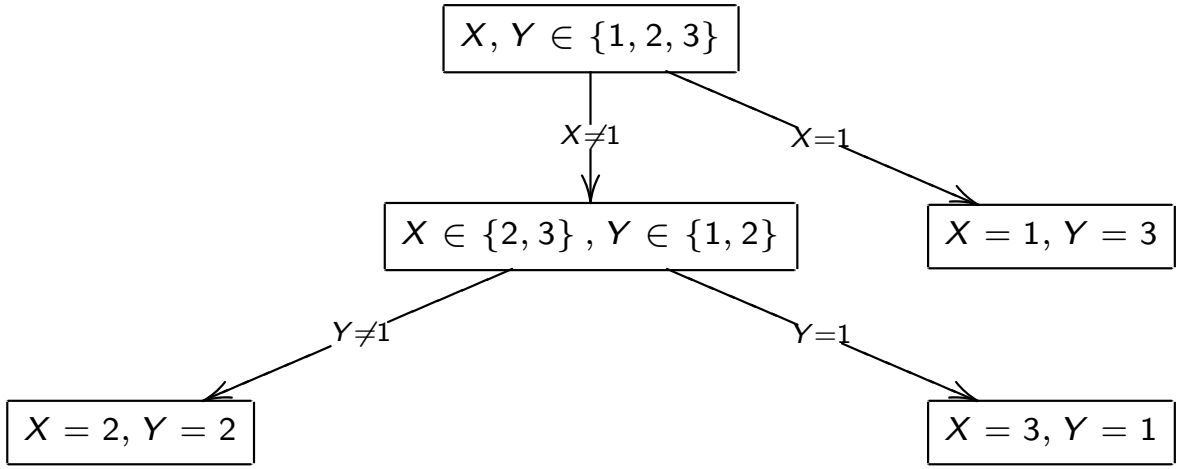


Példa: $X, Y \in \{1, 2, 3\}$ $X + Y = 4$

Felsorolásos disztribúciós stratégia

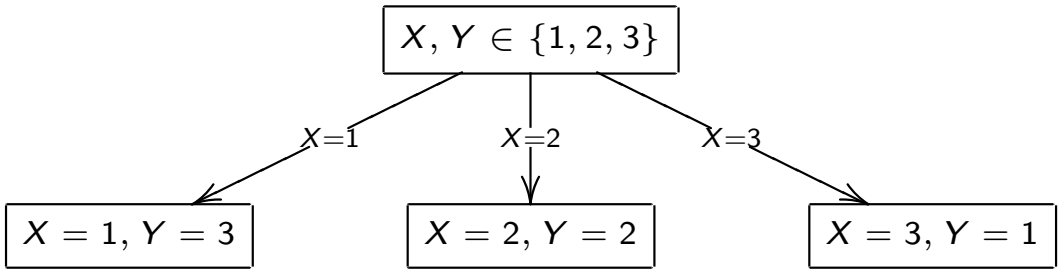


Lépegető disztribúciós stratégia, mélységi kereséssel

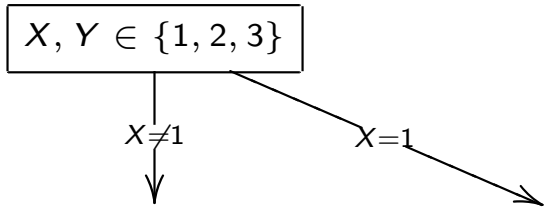


Példa: $X, Y \in \{1, 2, 3\}$ $X + Y = 4$

Felsorolásos disztribúciós stratégia

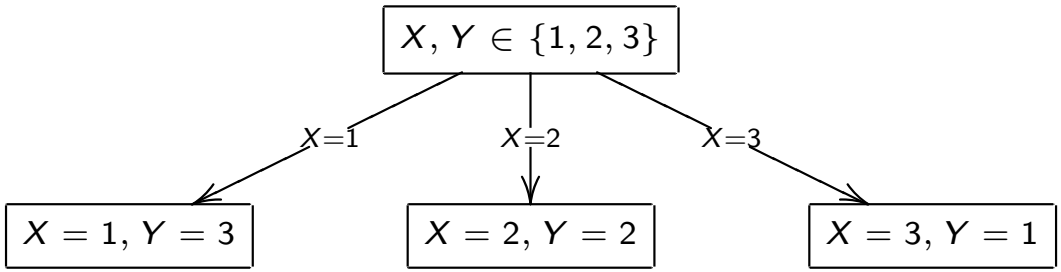


Lépegető disztribúciós stratégia, szélességi kereséssel

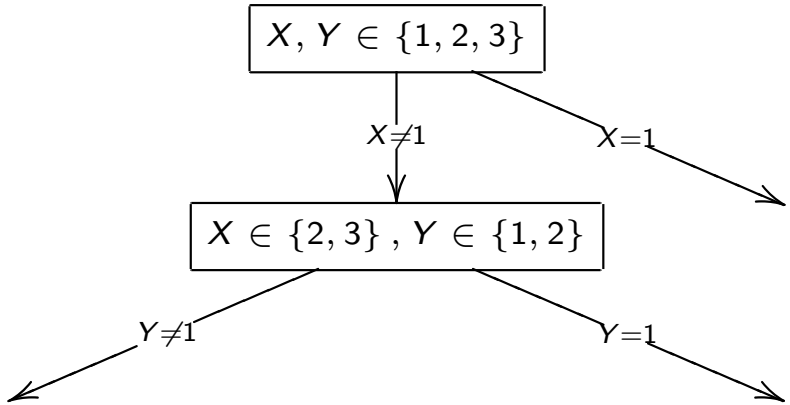


Példa: $X, Y \in \{1, 2, 3\}$ $X + Y = 4$

Felsorolásos disztribúciós stratégia

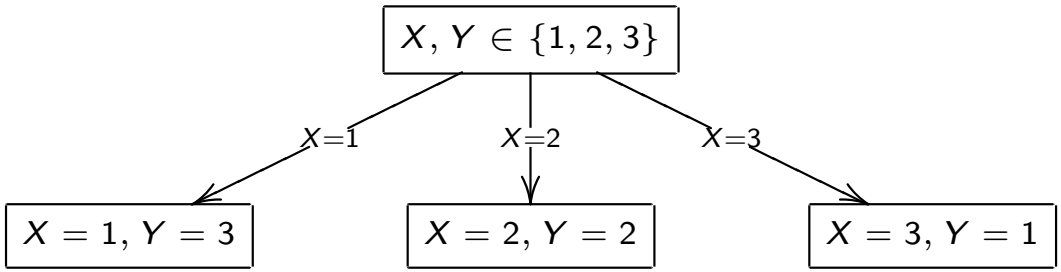


Lépegető disztribúciós stratégia, szélességi kereséssel

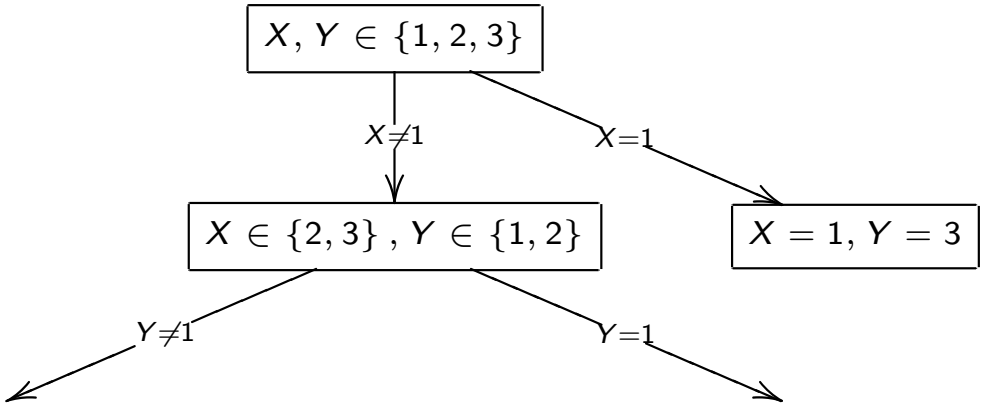


Példa: $X, Y \in \{1, 2, 3\}$ $X + Y = 4$

Felsorolásos disztribúciós stratégia

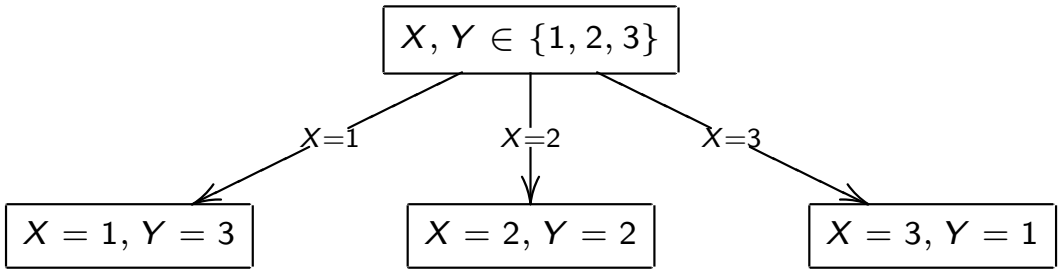


Lépegető disztribúciós stratégia, szélességi kereséssel

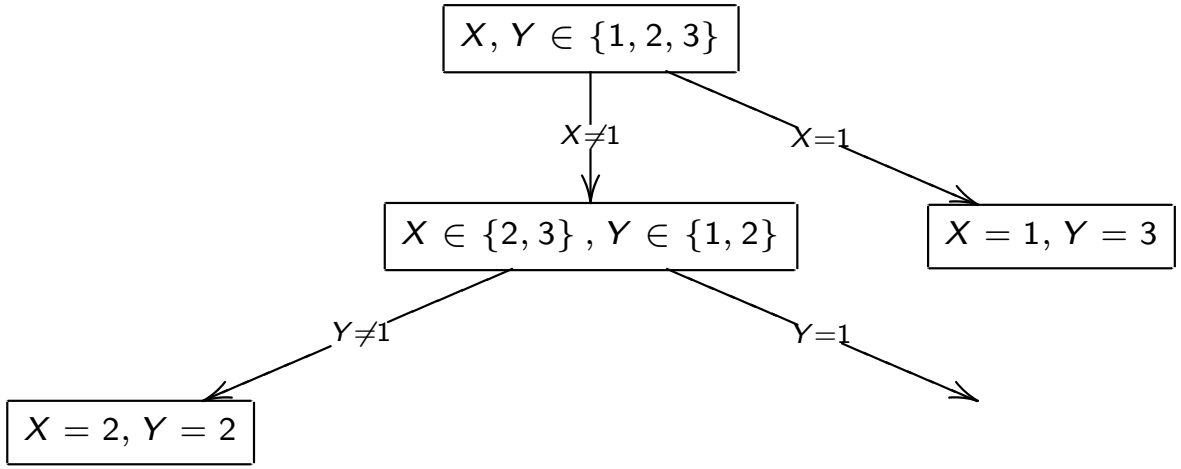


Példa: $X, Y \in \{1, 2, 3\}$ $X + Y = 4$

Felsorolásos disztribúciós stratégia

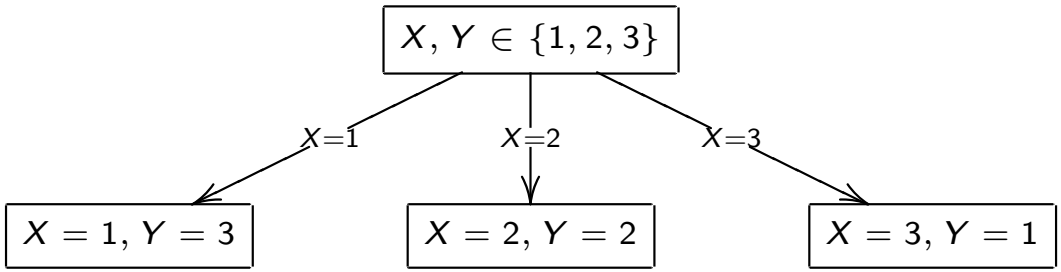


Lépegető disztribúciós stratégia, szélességi kereséssel

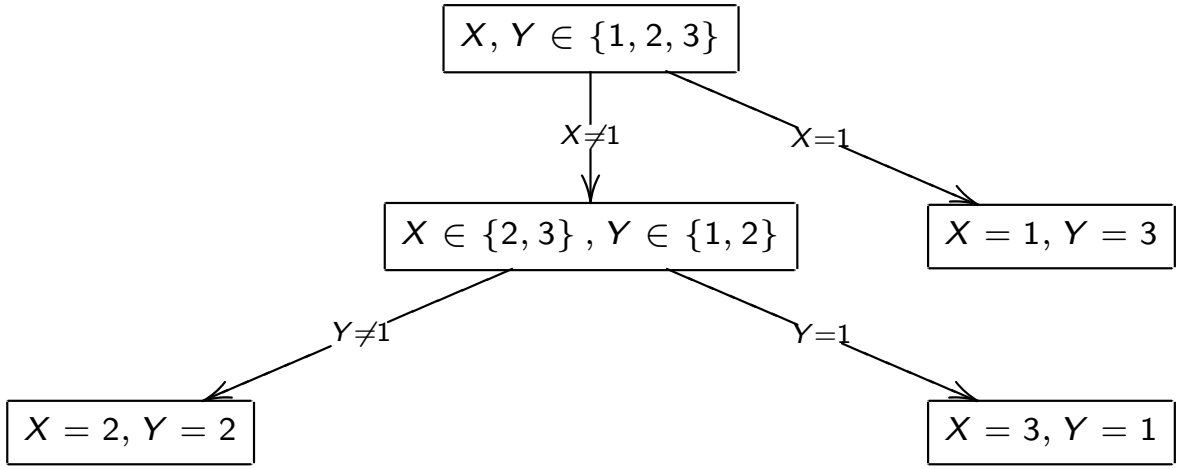


Példa: $X, Y \in \{1, 2, 3\}$ $X + Y = 4$

Felsorolásos disztribúciós stratégia



Lépegető disztribúciós stratégia, szélességi kereséssel



Számítási terek kezelése, a Space modul

`{NewSpace` *P X* `}` *X* egy újonnan létrehozott tér, amelyben a *P* unáris eljárás fut egy új változóval, a tér gyökér változójával, mint argumentummal meghívva.

`{Choose` *N X* `}` egy *N* alternatívás választási pont létrehozása. A szál blokkolódik, amíg a kereső függvény végre nem hajtja a választást, ekkor *X* a választott alternatíva sorszáma lesz.

`{Ask` *S X* `}` blokkolódik, amíg *S* stabil nem lesz, majd visszaadja *X*-ben a tér állapotát. Ennek lehetséges értékei:

- `failed`
- `merged`
- `succeeded`
- `alternatives(`*N*`)`

`{Clone` S_0 S_1 `}` blokkolódik, amíg S_0 stabil nem lesz, majd S_1-ben visszaadja egy másolatát.

`{Merge` S X `}` összefésüli az S teret a jelenlegivel, az X változót pedig egyesíti S gyökér változójával.

`{Commit` S N `}` blokkolódik, amíg S stabil nem lesz, majd a választási ponton kiválasztja az N. alternatívát. $N = L\#H$ esetén egy internvallumra szűkíti a lehetséges alternatívákat. (A `Choose` hívás akkor fog visszatérni az S térben, ha már csak egy alternatíva maradt.)

`{Inject` S P `}` egy új szálat indít az S térben, amely a P unáris eljárást fogja futtatni a tér gyökér változójával, mint argumentummal meghívva.

A beépített kereső eljárások koncepciója

- A Mozart `Search` moduljának kereső a feladatok leírását unáris procedúraként várják. Az argumentum a feladat megoldása, amit a létrehozott számítási tér gyökér változójával fognak egyesíteni.
- A visszatérési érték mindig a megoldások listája.
- Többféle keresés végezhető: csak az első megoldás előállítása, minden megoldás listázása vagy egy értékelő függvény szerinti optimális megoldás kiszámítása.
- Az értékelő függvénynek két argumentuma van, az első egy már megtalált megoldás, a második egy még folyamatban lévő keresés számítási terének gyökér változója. Az értékelő függvénynek az *„a gyökér változó jobb megoldás, mint az első argumentumban kapott”* kényszert kell kifejeznie.

`Search.base`: mélységi keresés

Eljárások:

`{Search.base.one` *Script Xs* `}`

`{Search.base.all` *Script Xs* `}`

`{Search.base.best` *Script Order Xs* `}`

A visszatérési érték a megoldások listája, vagy `nil`, ha nincs megoldás.

Általános kereső eljárások

`{Search.one.depth` *Script Recomp KillP Xs* `}`

`{Search.one.bound` *Script Bound Recomp KillP Xs* `}` a visszatérési érték `cut`, ha a keresés elérte a *Bound* mélységkorlátot.

`{Search.one.iter` *Script Recomp KillP Xs* `}`

`{Search.all` *Script Recomp KillP Xs* `}`

`{Search.best.bab` *Script Order Recomp KillP Xs* `}` branch and bound keresést végez: egy megoldás előállítása után a többi ágon felveszi az *Order* kényszert.

`{Search.best.restart` *Script Order Recomp KillP Xs* `}` egy megoldás előállítása után újrakezdi a keresést az *Order* kényszert is aklalmazva.

Az általános eljárások paraméterei:

KillP egy argumentumok nélküli eljárás, amit meghívva a keresés leállítható. A leállítás csak az elágazási pontok kifejtésére vonatkozik! A végtelen ciklusba esett szálakat nem állítja le.

Recomp az újraszámítási érték. A keresési fának csak minden *Recomp*-adik szintjét fogja a kereső a memóriában tartani, a köztes szinteket mindig a felettük lévőből számítja ki, ha szükség van rájuk. Az újraszámítás jó heurisztika mellett memóriát és időt is spórolhat!

Párhuzamos keresés

- Elosztott környezetben több hely összekapcsolásával elosztott keresést lehet végezni.
- Ha nagy az elágazási tényező és mély a keresési tér, a párhuzamos feldolgozás jelentősen gyorsít, bár van némi számítási overheadje ennek a keresésnek.
- A megoldandó feladat kódján nem kell módosítani! Csak egy funktorba kell ágyazni a szkriptet.
- A funktornak a következő eljárásokat kell exportálnia:

  `Script` a problémát leíró unáris eljárás.

  `Order` a rendező kényszer (csak optimum kereséséhez).

A `Search.parallel` osztály

Az alábbiakban F vagy egy funktor, vagy egy befőttbe írt funktorra mutató URL lehet.

`init(`$Host_1 : N_1 \# Fork_1 \ldots Host_n : N_n \# Fork_n$`)` egy új keresőmotor létrehozása a megadott gépeken. N az adott gépen indítandó Oz folyamatok száma.

`one(`$F\ Xs$`)`

`all(`$F\ Xs$`)`

`best(`$F\ Xs$`)`

`stop()`

`close()` leállítja a keresőmotort és a létrehozott távoli folyamatokat.

Választási pontok létrehozása

Alapvető szintaxis

```
case {Choose n}
of 1 then S1
[] 2 then S2
...
[] n then Sn
end
```

Édesített szintaxis

```
choice S1
[] S2
...
[] Sn
end
```

dis szerkezet

```
dis G1 then S1
[] G2 then S2
...
[] Gn then Sn
end
```

Ahol:

G_i : $X_1 \ldots X_m$ `in` S alakú őrfeltétel;

S_i : utasítás(sorozat);

`then` S : a `dis` szerkezetben a `then` ág elhagyható.

A `dis` szerkezet szemantikája

- A szál felfüggesztődik.
- Minden ágra létrejön egy-egy altér, egy-egy új szállal, amik az őrfeltételeket hajtják végre.
- A meghiúsult alterek kiesnek, és ha csak egy marad, akkor a megmaradt tér összefésülésre kerül a szülőével, és a szülő szál a megfelelő S_i végrehajtásával folytatódik.
- Egyébként választási pont jön létre a maradék terek között.

Az or szerkezet

Szintaktika:

Általános szerkezet

or G_1 then S_1
[] G_2 then S_2
...
[] G_n then S_n
end

Ahol:

- G_i : $X_1 \ldots X_m$ in S alakú őrfeltétel;
- S_i : utasítás(sorozat).

A then S ág elhagyható.

Szemantika:

- A szál felfüggesztődik.
- Minden ágra létrejön egy-egy altér, egy-egy új szállal, amik az őrfeltételeket hajtják végre.
- A szülő szál blokkolva marad, amíg legfeljebb egy kivételtől eltekintve az alterek meg nem hiúsulnak.
- Ha minden ág meghiúsult, kivétel váltódik ki, egyébként a megmaradt tér összefésülésre kerül a szülőével, és a szülő szál a megfelelő S_i végrehajtásával folytatódik.

A cond szerkezet

Szintaktika:

Általános szerkezet

```
cond G1 then S1
[] G2 then S2
...
else Sn
end
```

Ahol:

- G_i : $X_1 \ldots X_m$ `in` S alakú őrfeltétel;
- S_i : utasítás(sorozat).

Szemantika:

- A szál felfüggesztődik.
- Minden ágra létrejön egy-egy altér, egy-egy új szállal, amik az őrfeltételeket hajtják végre.
- Ha az egyik altér logikai következménye lesz a szülő térnek, a szülő szál feléled, és a megfelelő S_i végrehajtásával folytatódik.
- Ha minden altér meghiúsul, S_n kerül kiválasztásra.

A `cond` megfelel a Prolog P_1 `->` Q_1 `;` P_2 `->` Q_2 `;` ... Q_n szerkezetének, de a feltételek kiértékelési sorrendje tetszőleges!

Példa: Append Mozart módra

```
proc {Append Xs Ys Zs}
   dis
      Xs = nil Ys = Zs
   [] X Xr Zr in
      Xs = X|Xr Zs = X|Zr then
      {Append Xr Ys Zr}
   end
end
local X Y Z in
   {Browse X#Y#Z}
   {Append [1 2 3] [a b c] Z}
   {Append X Y [1 2 3 a b c]}
end
```

A feladat leírása

- A szerver program oldjon meg számrejtvényeket! Egy számrejtvény bemenete egész számok listája és egy egész érték, a cél. A megfejtést olyan kifejezések jelentik, amelyek a „+” „-” és „*” műveletekkel kapcsolják össze a listában szereplő számokat, és eredményük egyenlő a céllal.
- A kliens a szervernek ad fel véletlen számrejtvényeket, és kiírja a kapott megoldásokat.
- A szerver egy jegyet ajánl fel a megoldó algoritmusához, amit egy fájlba ír ki, hogy a kliens be tudja olvasni.