Konkurens programozás (folytatás)

Rövid ismétlés a tisztán funkcionális konkurenciáról

Konkurens programoknál több vezérlési szálat hozunk létre, amelynek viszont *nincs értelme* tisztán funkcionális programok esetében:

- Nincsenek (implicit) mellékhatások
- A kiértékelés sorrendje nem kötött

Ezért itt valójában mellékhatással rendelkező ("effectful") programrészek strukturálását, modularizálását valósítjuk meg.

Következmények:

- Az *IO* monád
- Nem determinisztikus viselkedés

Nem azonos a párhuzamos programozással, lásd hamarosan!

Holtpontok keletkezése

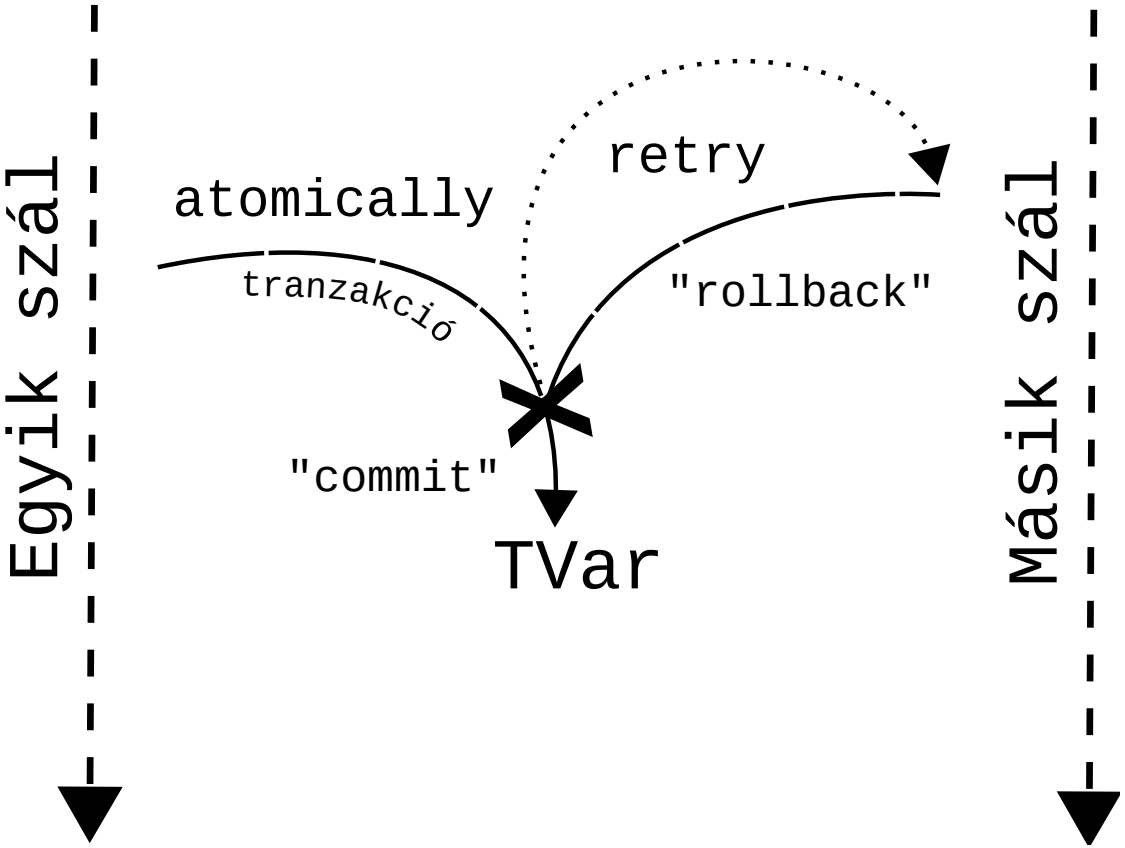
Az étkező filozófusok problémája itt is megjelenik:

```
type Dir α β = Map α (MVar (Set β))

moveElem :: (Ord α, Ord β) ⇒ Dir α β → β → α → α → IO ()
moveElem dir x a b = do
  let ma = dir ! a
  let mb = dir ! b
  va ← takeMVar ma
  vb ← takeMVar mb
  putMVar ma (Set.delete x va)
  putMVar mb (Set.insert x vb)

deadlockAction :: Int → Int → IO ()
deadlockAction n m = do
  ma ← newMVar (Set.fromList [1 ... n])
  mb ← newMVar (Set.fromList [(n + 1) ... (n + m)])
  let d = Map.fromList [('a', ma), ('b', mb)]
  forkIO $
    forM_ [1 ... n] $ λ i . moveElem d i 'a' 'b'
  forkIO $
    forM_ [(n + 1) ... (n + m)] $ λ i . moveElem d i 'b' 'a'
  readMVar ma >>= print
  readMVar mb >>= print
```

Szerencsére van nekünk... [3..21]

Az *STM* monád, *TVar*

TVar: Tranzakcionális védett változók, atomi blokkokban



Ez egy külön monád, mivel a típusrendszer így tud bizonyos megbízhatósági garanciákat nyújtani.

Az *STM* monád, *TVar* (példa)

Az előbbi, holtpontveszélyes kód *STM* segítségével leírva:

```
type TDir α β = Map α (TVar (Set β))

moveElem :: (Ord α, Ord β) ⇒ TDir α β → β → α → α → STM ()
moveElem dir x a b = do
  let ma = dir ! a
  let mb = dir ! b
  va ← readTVar ma
  vb ← readTVar mb
  writeTVar ma (Set.delete x va)
  writeTVar mb (Set.insert x vb)

atomicAction :: Int → Int → IO ()
atomicAction n m = do
  ma ← atomically $ newTVar $ Set.fromList [1 ... n]
  mb ← atomically $ newTVar $ Set.fromList [(n + 1) ... (n + m)]
  let d = Map.fromList [('a', ma), ('b', mb)]
  forkIO $
    forM_ [1..n] $ λ i . atomically $ moveElem d i 'a' 'b'
  forkIO $
    forM_ [(n + 1) ... (n + m)] $ λ i . atomically $ moveElem d i 'b' 'a'
  atomically (readTVar ma) >>= print
  atomically (readTVar mb) >>= print
```

Az *STM* monád: *MVar*

Az *MVar* kifejezhető az *STM* segítségével (de nem helyettesíti):

```
newtype TMVar α = TMVar (TVar (Maybe α))

newEmptyTMVar :: STM (TMVar α)
newEmptyTMVar = do
  t ← newTVar Nothing
  return (TMVar t)
takeTMVar :: TMVar α → STM α
takeTMVar (TMVar t) = do
  m ← readTVar t
  case m of
    Just x → do
      writeTVar t Nothing
      return x
    _ → retry
```

Kompozicionalitás:

```
takeTwoTMVars = atomically $ do
  a ← takeTMVar ta
  b ← takeTMVar tb
  return (a, b))
```

Az STM monád: *MVar* (folytatás)

```
putTMVar :: TMVar α → α → STM ()
putTMVar (TMVar t) x = do
  m ← readTVar t
  case m of
    Nothing → do
      writeTVar t (Just x)
      return ()
    _ → retry

-- orElse :: STM α → STM α → STM α
takeEitherTMVar :: TMVar α → TMVar β → STM (Either α β)
takeEitherTMVar ma mb =
  (do
    x ← takeTMVar ma
    return (Left x))
  `orElse`
  (do
    x ← takeTMVar mb
    return (Right x))
```

orElse: Két blokkoló tranzakció közül valamelyik, de nem mind a kettőnek, elvégzése.

Az STM monád (példa)

```
data Task α = Process α | Stop

concurrentMapM :: Int → (α → IO β) → [α] → IO [β]
concurrentMapM k f xs = do
  processed ← newTChanIO
  jobs      ← newTChanIO
  workers   ← newTVarIO k
  replicateM_ k $ forkIO $ do
    worker jobs processed
    atomically $ modifyTVar workers $ subtract 1
  forM_ xs $ λ i . atomically $ writeTChan jobs $ Process i
  replicateM_ k $ atomically $ writeTChan jobs Stop
  result ← forM xs $ λ _ . atomically $ readTChan processed
  atomically $ readTVar workers >>= (λ w . check $ w ≡ 0)
  return $ concat result
  where
    worker xs ys = do
      job ← atomically $ readTChan xs
      case job of
        Stop      → return []
        Process x → do
          y ← f x
          atomically $ writeTChan ys [y]
          worker xs ys
```

Az *STM* korlátai

- *STM*: Kompozicionális atomiság, kompozicionális blokkolás, egyszerűbb hibakezelés.
- Az *MVar* viszont gyorsabb (tud lenni)!
- Az *MVar* viszonylag pártatlan ("fair"); az *STM* ellenben nem igazán alkalmas szálak közti többirányú kommunikáció modellezésére. A kompozicionalitás a pártatlanság ellen hat. Vessük össze:

  $atomically\ \$\ forM\ ts\ \$\ \lambda\ t\ .\ takeTMVar\ t$

  kontra

  $forM\ ts\ \$\ \lambda\ t\ .\ atomically\ \$\ takeTMVar\ t$

Az *STM* korlátai: teljesítmény

Az *STM* költségének megértéséhez az implementáció rövid áttekintése:

- Az *STM* tranzakciók futtatásakor egy naplóban tároljuk a menet közben elvégzett *readTVar* és *writeTVar* műveleteket.
- Emiatt a visszagörgetéskor egyszerűen csak el kell dobni a naplót ($O(1)$).
- A *readTVar* esetén mindig ellenőrizni kell, hogy az adott változó módosult-e ($O(n)$).
- Amikor elérjük a tranzakció végét, akkor a *readTVar* megvizsgálja, hogy semelyik változó nem változott. Ha nem, akkor véglesítődik az eredmény, ellenben visszagörgetjük.

Ezért: érdemes kerülni a hosszú részműveleteket és a sok *TVar* értéket tartalmazó tranzakciókat.

Párhuzamos programozás

Mitől párhuzamos egy program?

- A *párhuzamos* programozás a számítás gyorsítására törekszik minél több feldolgozó egység jobb kihasználásával.
- A párhuzamosság lehetőség szerint *determinisztikus*: a viselkedés teljesen megegyezik a szekvenciális változattal, csupán gyorsabban fog működni.
- A programok lehetnek egyszerre párhuzamosak és konkurensek is.

Párhuzamosság tisztán funkcionális esetben

- A tisztán funkcionális programok párhuzamosítása nem automatikus, de gyakran egyszerűbb.
  - Probléma: egy rosszul párhuzamosított program lassabb is lehet, mint annak szekvenciális változata.
  - Nincsenek igazán jól bevált automatikus eszközök.
- Magasszintű, deklaratív eszközök (annotációk), intelligensebb futtató rendszer.
  - A szinkronizáció és kommunikáció nem explicit.
- Hátrány: a teljesítménybeli problémák felderítése és javítása nehezebb.
  - Kulcs: a teljesítmény értékelhető mérése és elemezhetősége.
- Fontos, miként bontjuk fel a problémát:
  - szemcsézettség (granularity)
  - adatfüggőségek (data dependencies)

Előkészítés: A lusta kiértékelés

```
> let x = 1 + 2 :: Int
> let y = x + 1

> :sprint x y
x = _
y = _
> -- x, y: "thunk"

> :type seq
seq :: a -> b -> b
> seq y ()
()

> :sprint x y
x = 3
y = 4
```

Előkészítés: Weak Head Normal Form (WHNF)

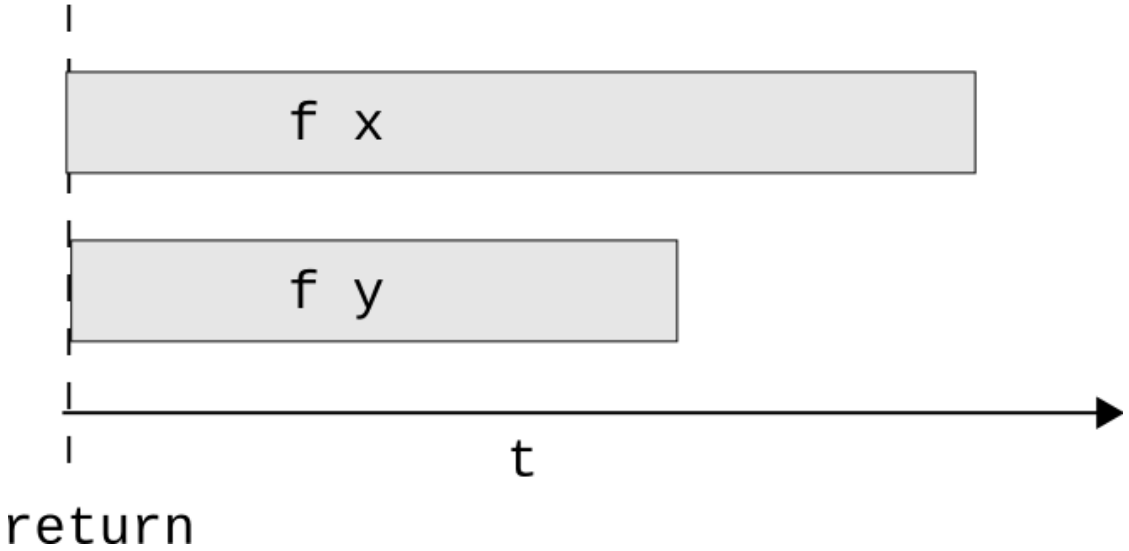
```
> import Data.Tuple (swap)
> let z = swap (x, x + 1)

> :sprint z
z = _

> seq z ()
()

> :sprint z
z = (_,_)
> -- z: weak head normal form

> seq x ()
()

> :sprint z
z = (_,3)
```

Előkészítés: Weak Head Normal Form (WHNF)

```
> let xs = map (+1) [1..10] :: [Int]
> :sprint xs
xs = _

> seq xs ()
()
> :sprint xs
xs = _ : _

> length xs
10
> :sprint xs
xs = [_,_,_,_,_,_,_,_,_,_]

> sum xs
65
> :sprint xs
xs = [2,3,4,5,6,7,8,9,10,11]
```

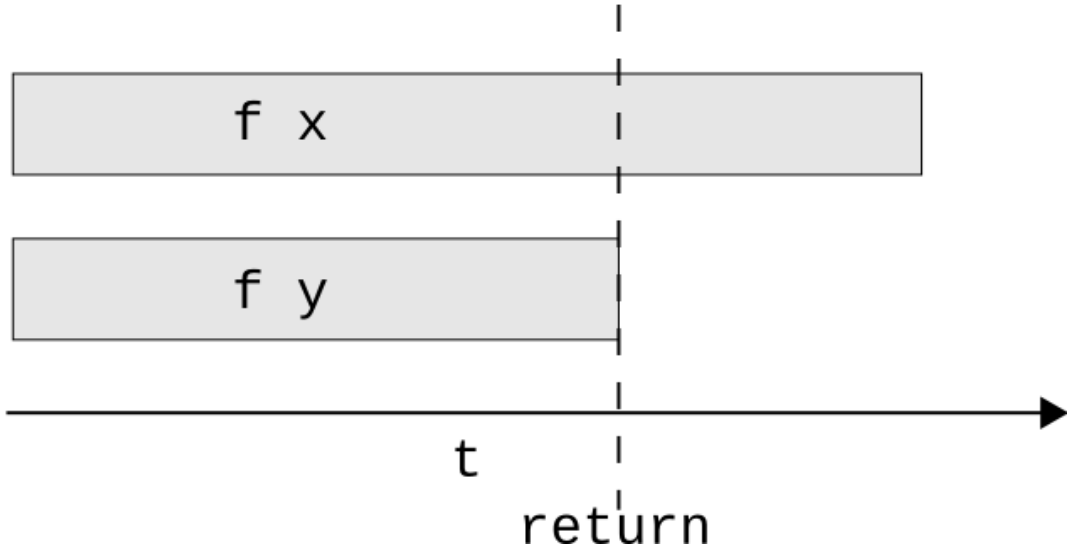
Az *Eval* monád: *rpar*/*rpar*

```
-- cabal install parallel
import Control.Parallel.Strategies

-- Van "run" függvény, nincs IO
parParAction f x y = runEval $ do
  x1 ← rpar (f x) -- WHNF párhuzamosan
  y1 ← rpar (f y) -- thunk (másképp nincs hatása)
  return (x1, y1)
```



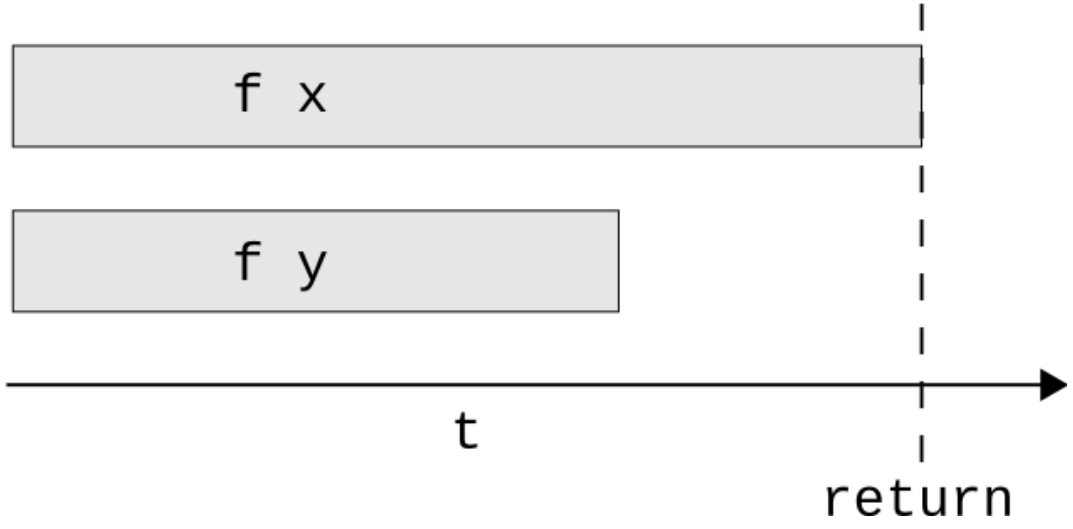
Az *Eval* monád: *rpar*/*rseq*

$$
\begin{aligned}
&\mathit{parSeqAction}\ f\ x\ y\ =\ \mathit{runEval}\ \$\ \mathbf{do}\\
&\quad x_1\ \leftarrow\ \mathit{rpar}\ (f\ x)\\
&\quad y_1\ \leftarrow\ \mathit{rseq}\ (f\ y)\ \text{-- megvárja a WHNF-et}\\
&\quad \mathit{return}\ (x_1,\ y_1)
\end{aligned}
$$



Az *Eval* monád: *rpar*/*rpar*/*rseq*/*rseq*

$$
\begin{aligned}
&\mathit{parParSeqSeqAction}\ f\ x\ y\ =\ \mathit{runEval}\ \$\ \mathbf{do}\\
&\quad x_1 \leftarrow \mathit{rpar}\ (f\ x)\\
&\quad y_1 \leftarrow \mathit{rpar}\ (f\ y)\\
&\quad \mathit{rseq}\ x_1\\
&\quad \mathit{rseq}\ y_1\\
&\quad \mathit{return}\ (x_1,\ y_1)
\end{aligned}
$$



Esettanulmány: Egy Sudoku megoldó

```
-- http://www.haskell.org/haskellwiki/Sudoku
import Sudoku
import System.Environment

main :: IO ()
main = do
  [f]  ← getArgs
  file ← readFile f
  let puzzles   = lines file
  let solutions = map solve puzzles
  print $ length [ s | Just s ← solutions ]
```

Esettanulmány: Egy Sudoku megoldó (profil)

```
$ ghc -O Solver1.hs -rtsopts -main-is Solver1
...
$ ./Solver1 sudoku17.1000.txt +RTS -s
1000
   2,362,886,904 bytes allocated in the heap
      38,757,536 bytes copied during GC
         239,376 bytes maximum residency (15 sample(s))
          81,232 bytes maximum slop
               2 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      4573 colls,     0 par    1.56s    0.06s     0.0000s    0.0004s
  Gen  1        15 colls,     0 par    0.01s    0.00s     0.0002s    0.0005s

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    0.01s  (  1.52s elapsed)
  GC      time    1.57s  (  0.06s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    1.58s  (  1.58s elapsed)

  %GC     time      99.2%  (4.1% elapsed)

  Alloc rate    186,730,433,380 bytes per MUT second

  Productivity   0.8% of total user, 0.8% of total elapsed
```