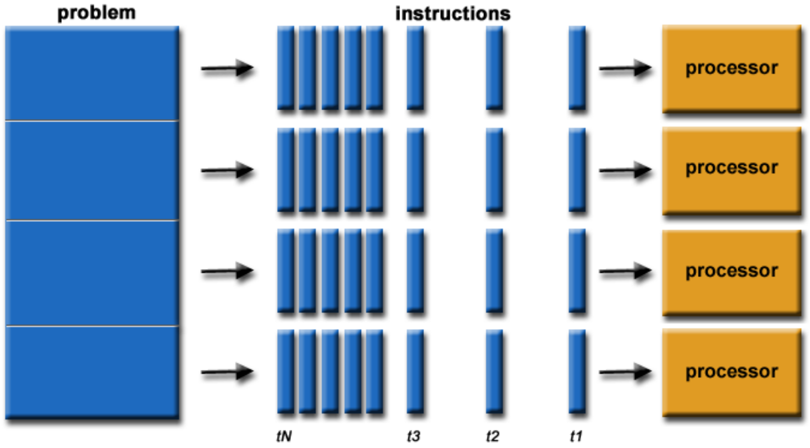
IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Paralelism in Haskell

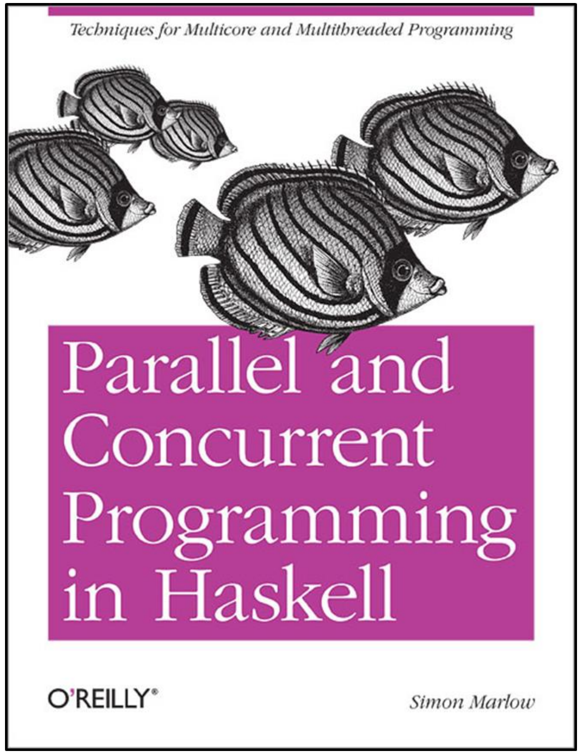
introducere

Ioana Leustean



sursa imaginii

Haskell



Chapter 2: Basic Parallelism

Parallelism is about speeding up a program by using multiple processors.

Pure Parallelism (Control.Parallel): Speeding up a pure computation using multiple processors.

Pure parallelism has these advantages:

- Guaranteed deterministic (same result every time)
- no race conditions or deadlocks.

Concurrency (Control.Concurrent): multiple threads of control that execute "at the same time".

- Threads are in the IO monad.
- IO operations from multiple threads are interleaved non-deterministically
- communication between threads must be explicitly programmed.
- Threads may execute on multiple processors simultaneously.
- Danger: race conditions and deadlocks.

https://wiki.haskell.org/Parallelism

- Paralelismul in Haskell se obtine folosind monada Eval care este definite in modulul Control.Parallel.Strategies

```
data Eval a
instance Monad Eval

runEval :: Eval a -> a

rpar :: a -> Eval a

rseq :: a -> Eval a
```

argumentul poate fi evaluat in paralel

evalueaza argumentul si asteapta rezultatul

```
import Control.Parallel.Strategies

-- fib  n   Fibonacci

test = runEval $ do
            x <- rpar (fib 36)
            y <- rpar (fib 37)
            z <- rseq  (fib  33)
            return (x,y,z)

main = print test
```

```
*Main> :! ghc --make  mypar1.hs
[1 of 1] Compiling Main             ( mypar1.hs, mypar1.o )
Linking mypar1.exe ...
```

mypar1.hs

```haskell
import Control.Parallel.Strategies

-- fib  n   Fibonacci

test = runEval $ do
         x <- rpar (fib 36)
         y <- rpar (fib 37)
         z <- rseq  (fib  33)
         return (x,y,z)

main = print test
```

```
D:\DIR\HS\myexc>mypar1 +RTS -s
(24157817,39088169,5702887)
  17,229,615,344 bytes allocated in the heap
      21,906,856 bytes copied during GC
          60,008 bytes maximum residency (4 sample(s))
          22,080 bytes maximum slop
               2 MB total memory in use (0 MB lost due to fragmentation)

                                    Tot time (elapsed)  Avg pause  Max pause
  Gen  0     32936 colls,     0 par    0.11s    0.15s     0.0000s    0.0009s
  Gen  1         4 colls,     0 par    0.00s    0.00s     0.0001s    0.0002s

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    8.24s  (  8.28s elapsed)
  GC      time    0.11s  (  0.15s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    8.36s  (  8.43s elapsed)

  %GC     time       1.3%  (1.7% elapsed)

  Alloc rate    2,091,771,670 bytes per MUT second

  Productivity  98.7% of total user, 97.9% o
```

CPU time: 8.36s/ wall-clock time: 8.43s

> mypar1 +RTS –s

Total time 8.36s (8.43s elapsed)

```
*Main> :! ghc --make  -threaded mypar1t.hs
[1 of 1] Compiling Main             ( mypar1t.hs, mypar1t.o )
Linking mypar1t.exe ...
```

mypar1t.hs

```
import Control.Parallel.Strategies

-- fib  n   Fibonacci

test = runEval $ do
         x <- rpar (fib 36)
         y <- rpar (fib 37)
         z <- rseq  (fib  33)
         return (x,y,z)

main = print test
```

```
D:\DIR\HS\myexc>mypar1t +RTS -N2 -s
(24157817,39088169,5702887)
  17,229,689,904 bytes allocated in the heap
      22,353,560 bytes copied during GC
         128,216 bytes maximum residency (4 sample(s))
          30,296 bytes maximum slop
               3 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     21779 colls, 21779 par    0.17s    0.14s     0.0000s    0.0093s
  Gen  1         4 colls,     3 par    0.00s    0.00s     0.0001s    0.0002s

  Parallel GC work balance: 42.83% (serial 0%, perfect 100%)

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N2)

  SPARKS: 2 (2 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    9.28s  (  5.87s elapsed)
  GC      time    0.17s  (  0.14s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    9.45s  (  6.01s elapsed)

                                    ytes per MUT second
```

> mypar1 +RTS -N2 –s

SPARKS: 2 (2 converted,…)

Total time 9.45 s (6.01s elapsed)

converted = folosite efectiv in paralelism

speedup 8.43/6.01 = 1.4

- Sparks

"The argument to rpar is called a spark. The runtime collects sparks in a pool and uses this as a source of work when there are spare processors available, using a technique called work stealing. Sparks may be evaluated at some point in the future, or they might not—it all depends on whether there is a spare core available. Sparks are very cheap to create: rpar essentially just writes a pointer to the expression into an array."

http://chimera.labs.oreilly.com/books/1230000000929/ch02.html#sec_par-eval-sudoku2

"Sparks are specific to parallel Haskell. Abstractly, a spark is a pure computation which may be evaluated in parallel. Whether or not a spark is evaluated in parallel with other computations, or other Haskell IO threads, depends on what your hardware supports and on how your program is written. Sparks are put in a work queue and when a CPU core is idle, it can execute a spark by taking one from the work queue and evaluating it.

On a multi-core machine, both threads and sparks can be used to achieve parallelism. Threads give you concurrent, non-deterministic parallelism, while sparks give you **pure deterministic parallelism**. Haskell threads are ideal for applications like network servers where you need to do lots of I/O and using concurrency fits the nature of the problem. Sparks are ideal for speeding up pure calculations where adding non-deterministic concurrency would just make things more complicated. "
https://wiki.haskell.org/Parallel/Glossary

Here are some other things that can happen to a spark:

converted

Are turned into real parallelism at runtime

dud

When rpar is applied to an expression that is already evaluated, this is counted as a dud and the rpar is ignored.

GC'd

The sparked expression was found to be unused by the program, so the runtime removed the spark. We'll discuss this in more detail in "GC'd Sparks and Speculative Parallelism".

fizzled

The expression was unevaluated at the time it was sparked but was later evaluated independently by the program. Fizzled sparks are removed from the spark pool.

"It is possible that a spark in the spark pool can refer to a computation that has already been evaluated by the program. Perhaps there were not enough processors to evaluate the spark in parallel, and another thread ended up evaluating the computation during the normal course of computing its results.
When a spark in the spark pool refers to a value, rather than an unevaluated computation, we say the spark has fizzled; this potential for parallel execution has expired"

Seq no more: Better Strategies for Parallel Haskel, 2010
http://research.microsoft.com/pubs/138042/haskell18-marlow.pdf

http://chimera.labs.oreilly.com/books/1230000000929/ch02.html#sec_par-eval-sudoku2

```haskell
import Control.Parallel.Strategies
import System.Environment

fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

test n1 n2 n3 = runEval $ do
   x <- rpar (fib n1)
   y <- rpar (fib n2)
   z <- rseq (fib n3)
   return (x,y,z)


main =  do
       [s1,s2,s3] <- getArgs
       let n1 = (read s1)::Integer
           n2 = (read s2):: Integer
           n3 = (read s3)::Integer
       print (test n1 n2 n3)
```

```
Prelude> :m + System.Environment
Prelude System.Environment> :t getArgs
getArgs :: IO [String]
```

```
*Main> :! ghc --make  -threaded mypar2t.hs
[1 of 1] Compiling Main             ( mypar2t.hs,
Linking mypar2t.exe ...
```

```
> mypar2t 25 30 10 +RTS –N2 -s
```

```
D:\DIR\HS\myexc>mypar2t 25 30 10 +RTS -N2 -s
(121393,1346269,89)
     366,926,856 bytes allocated in the heap
         476,568 bytes copied during GC
          94,792 bytes maximum residency (2 sample(s))
          26,192 bytes maximum slop
               2 MB total memory in use (0 MB lost due to fragmenta

                                    Tot time (elapsed)  Avg pause
  Gen  0       642 colls,   642 par    0.03s    0.03s     0.0000s
  Gen  1         2 colls,     1 par    0.00s    0.00s     0.0001s

  Parallel GC work balance: 8.43% (serial 0%, perfect 100%)

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N2)

  SPARKS: 2 (1 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    0.16s  (  0.16s elapsed)
  GC      time    0.03s  (  0.03s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    0.20s  (  0.19s elapsed)
```

- mai mult paralelism

```
myparMap :: (a -> b) -> [a] -> Eval [b]

myparMap f [] = return []
myparMap f (a:as) = do
      b <- rpar (f a)
      bs <- myparMap f as
      return (b:bs)
```

```
import Control.Parallel.Strategies
--import System.Environment




arg =[34,35,25, 27]

fib :: Integer -> Integer
fib 0 = 1fib 1 = 1
fib n = fib (n-1) + fib (n-2)

myparMap :: (a -> b) -> [a] -> Eval [b]
myparMap f [] = return []
myparMap f (a:as) = do
        b <- rpar (f a)
        bs <- myparMap f as
        return (b:bs)

test  = runEval (myparMap fib arg)

main =  print test
```

➢ rpar si rseq

rpar	"my argument can be created in parallel"
rseq	" evaluate my argument and wait for the result"

```
test = runEval $ do
         x <- rpar (fib 36)
         y <- rpar (fib 37)
         return (x,y)
```

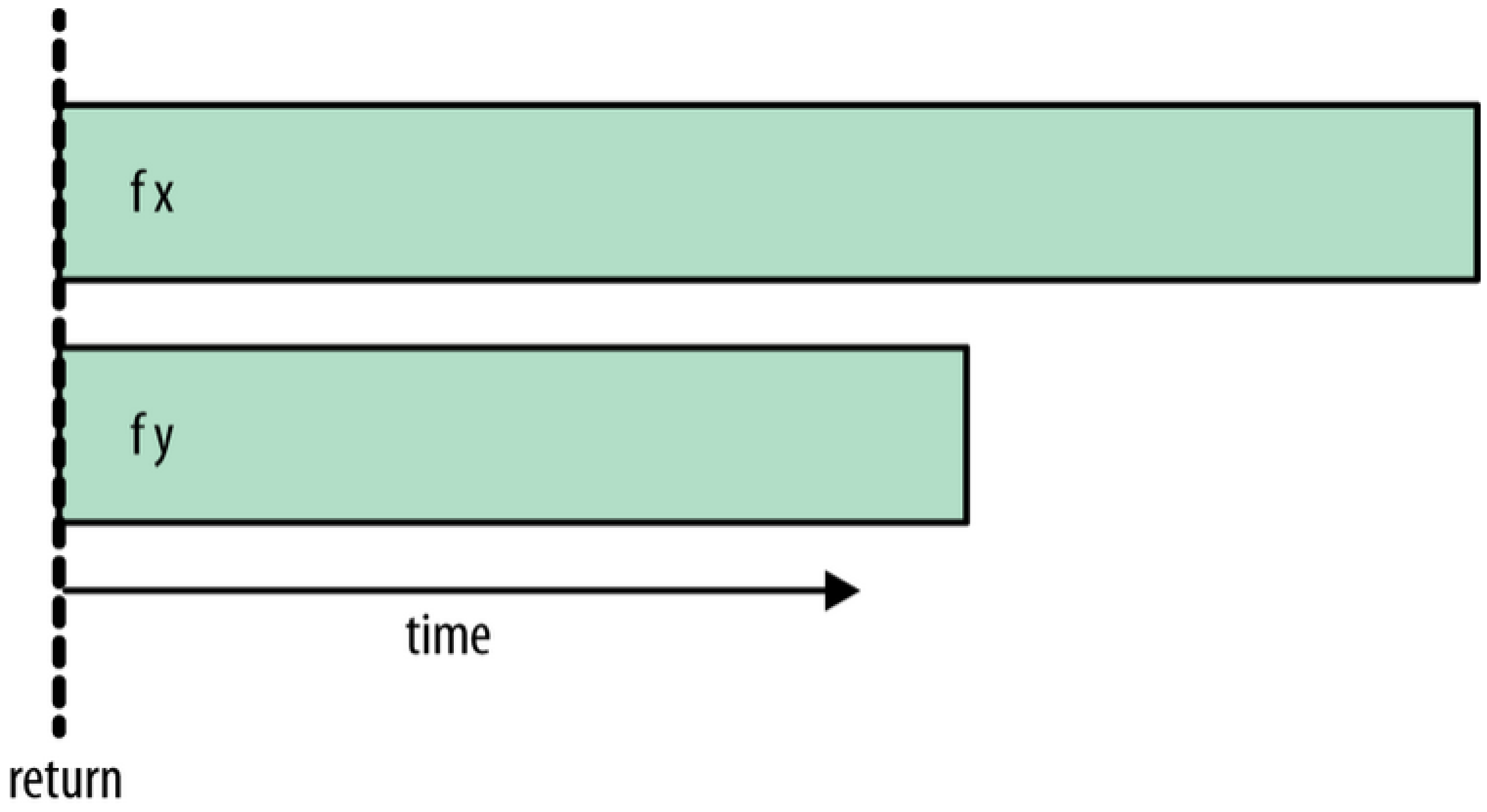


Figure 2-5. rpar/rpar timeline

http://chimera.labs.oreilly.com/books/1230000000929/ch02.html#sec_par-rpar-rseq

➢ rpar si rseq

rpar "my argument can be created in parallel"
rseq " evaluate my argument and wait for the result"

```
test = runEval $ do
          x <- rpar (fib 37)
          y <- rseq (fib 36)
          return (x,y)
```

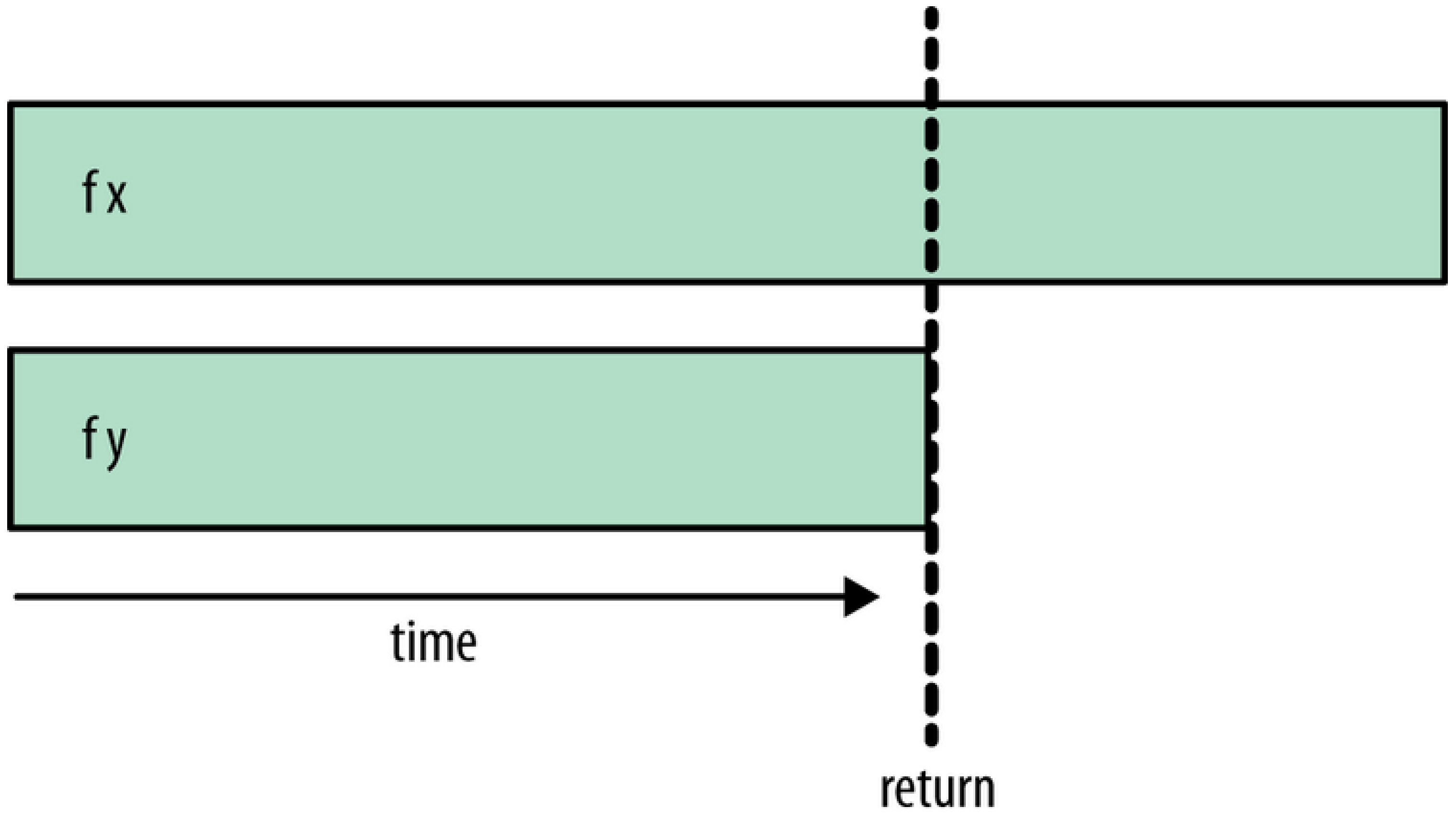


Figure 2-6. rpar/rseq timeline

nu e indicata, deoarece nu stim care din operatii se termina prima!

http://chimera.labs.oreilly.com/books/1230000000929/ch02.html#sec_par-rpar-rseq

➢ rpar si rseq

rpar	"my argument can be created in parallel"
rseq	" evaluate my argument and wait for the result"

```
test = runEval $ do
         x <- rpar (fib 37)
         y <- rpar (fib 36)
         rseq x
         rseq y
         return (x,y)
```

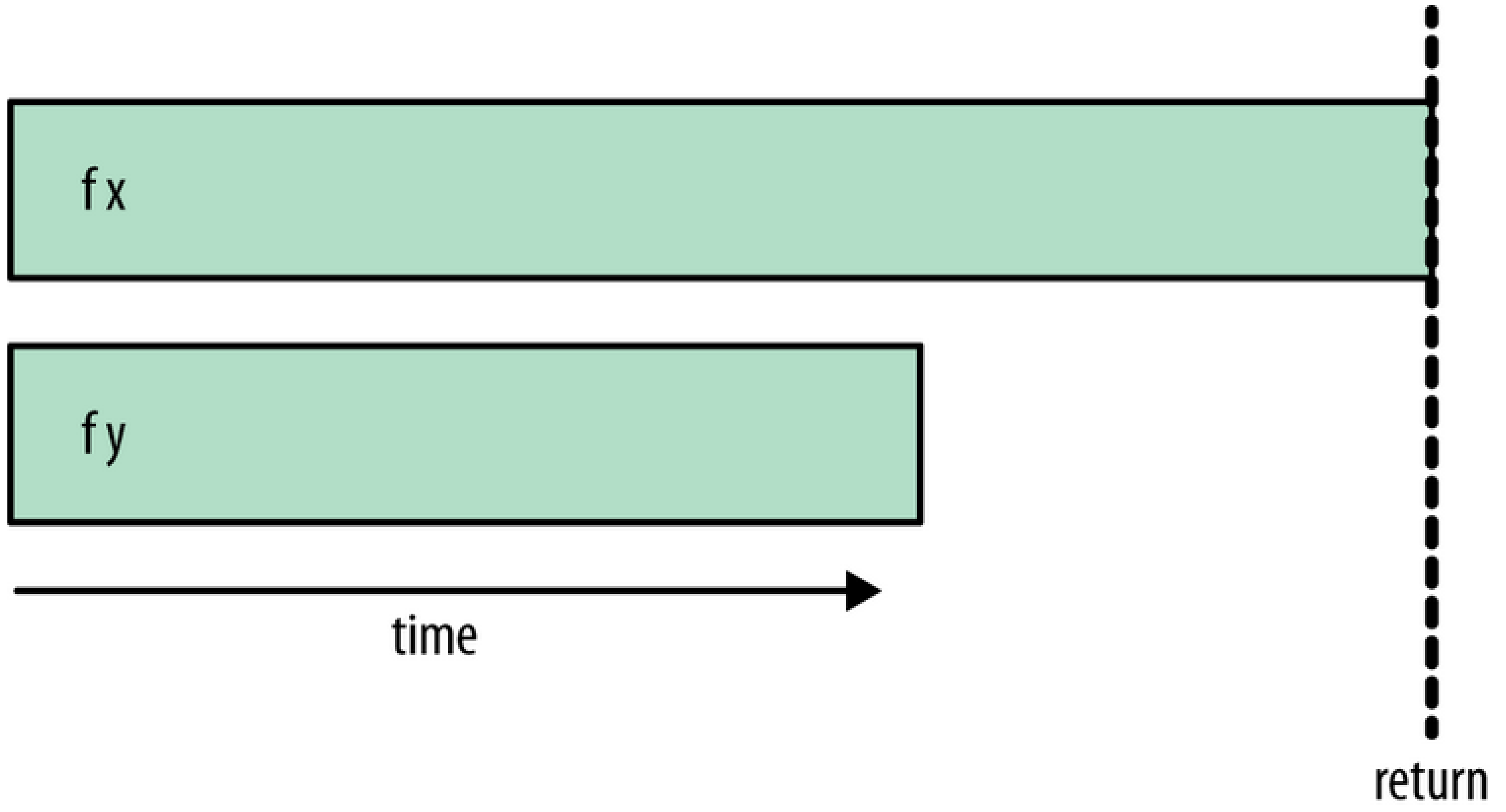


Figure 2-7. rpar/rseq/rseq timeline

(la fel pt. *rpar/rpar/rseq/rseq*)

http://chimera.labs.oreilly.com/books/1230000000929/ch02.html#sec_par-rpar-rseq

- Thunk (o expresie neevaluata)
- WHNF (week head normal form)

```
Prelude> let x = 1+2 :: Int
Prelude> :sprint x
x = _
Prelude> let y =(x,x)
Prelude> :sprint y
y = _
Prelude> :m + Control.Exception
Prelude Control.Exception> ye <- evaluate (x,x)
Prelude Control.Exception> :sprint ye
ye = (_,_)
Prelude Control.Exception> xe <-evaluate(x)
Prelude Control.Exception> :sprint xe
xe = 3
```

:sprint scrie valoarea unei expresii fara a-I forta evaluarea

```
Prelude Control.Exception> :t evaluate
evaluate :: a -> IO a
```

evaluate isi evalueaza argumentul la whnf atunci cand actiunea IO este executata

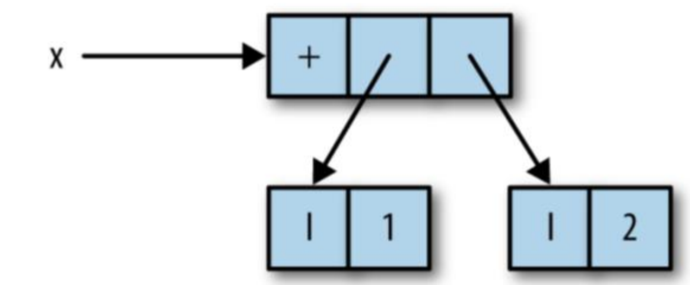


Figure 2-1. The thunk representing 1 + 2

x este un "thunk"

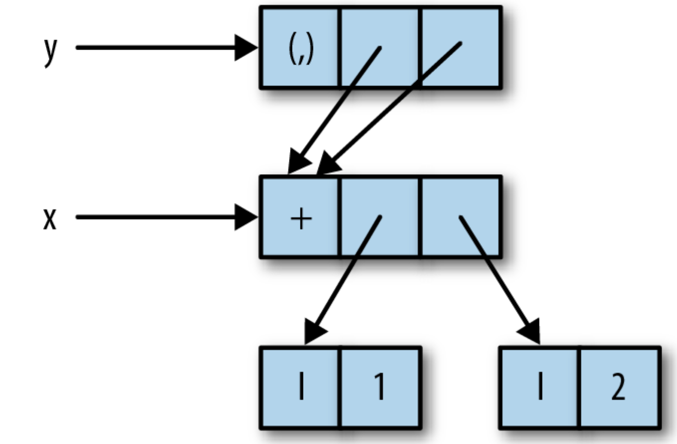
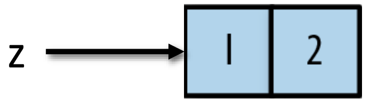


Figure 2-3. A pair with both components referring to the same thunk

y este o "whnf"



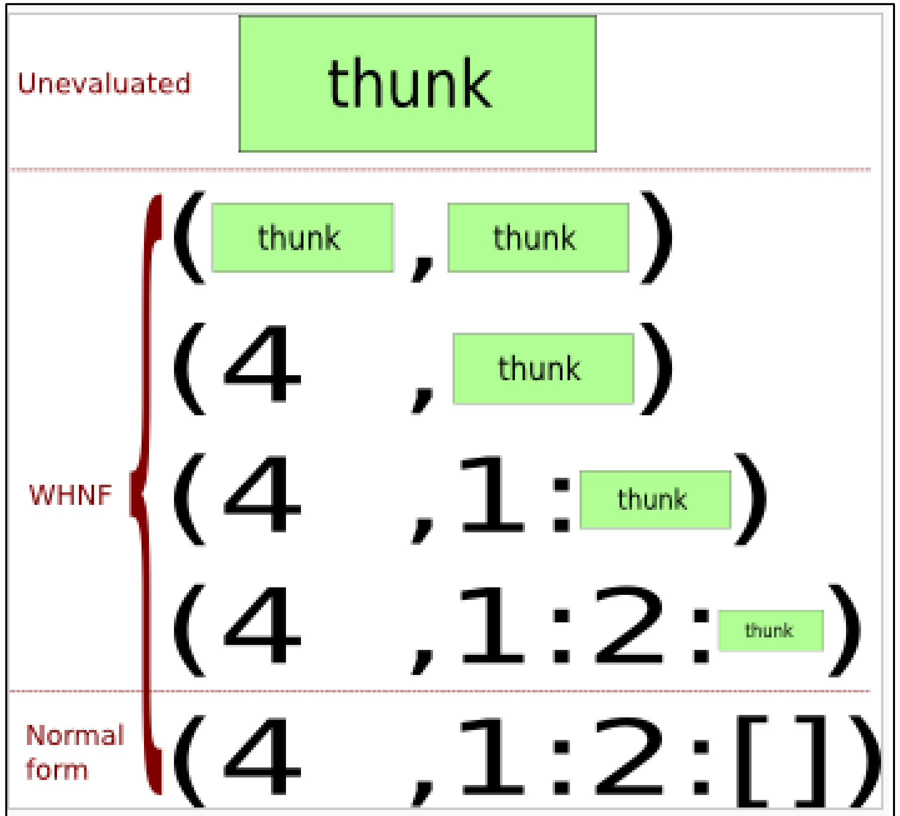
z este o forma normala

Thunk. WHNF

- Un thunk este o expresie neevaluata
- O expresie este in whnf daca, in reprezentarea grafica, radacina arborelui este un constructor

  + 1 2 nu este whnf
  3: x:[] este whnf

- O expresie este in forma normal daca este complet evaluata.



Evaluating the value `(4, [1, 2])` step by step. The first stage is completely unevaluated; all subsequent forms are in WHNF, and the last one is also in normal form.

http://en.wikibooks.org/wiki/Haskell/Laziness#Thunks_and_Weak_head_normal_form

➢ WHNF

rpar si rseq isi evalueaza argumentele la whnf

```
Prelude Control.Exception Control.Parallel.Strategies> yer <- evaluate $ runEval (rpar (x,x))
Prelude Control.Exception Control.Parallel.Strategies> :sprint yer
yer = (_,_)
Prelude Control.Exception Control.Parallel.Strategies> yeq <- evaluate $ runEval (rseq (x,x))
Prelude Control.Exception Control.Parallel.Strategies> :sprint yeq
yeq = (_,_)
```

```
Prelude Control.Exception Control.Parallel.Strategies> let ls = map (+1)[1,2,3]
Prelude Control.Exception Control.Parallel.Strategies> :sprint ls
ls = _
Prelude Control.Exception Control.Parallel.Strategies> lse <- evaluate $ runEval (rpar ls)
Prelude Control.Exception Control.Parallel.Strategies> :sprint lse
lse = _ : _
```

- force

rpar si rseq isi evalueaza argumentele la whnf

```
Prelude Control.Exception Control.Parallel.Strategies> let ls = map (+1)[1,2,3]
Prelude Control.Exception Control.Parallel.Strategies> :sprint ls
ls = _
Prelude Control.Exception Control.Parallel.Strategies> lse <- evaluate $ runEval (rpar ls)
Prelude Control.Exception Control.Parallel.Strategies> :sprint lse
lse = _ : _
```

Daca dorim ca ls sa fie complet evaluate de rpar atunci putem sa "fortam" evaluarea folosing functia force

```
import Control.DeepSeq
force :: NFData a => a -> a
```

```
> lse1 <- evaluate $ runEval (rpar (force ls))
> :sprint lse1
> [2,3,4]
```

```
Prelude Control.Exception Control.Parallel.Strategies> :m + Control.DeepSeq
Prelude Control.Exception Control.Parallel.Strategies Control.DeepSeq> lse1 <- evaluate $ runEval (rpar (force ls))
Prelude Control.Exception Control.Parallel.Strategies Control.DeepSeq> :sprint lse1
lse1 = [2,3,4]
Prelude Control.Exception Control.Parallel.Strategies Control.DeepSeq> :t force
force :: NFData a => a -> a
```

➢ Strategii de evaluare

```
*Main> :t rpar
rpar :: a -> Eval a
```

rpar/rpar poate fi vazuta ca o strategie de evaluare

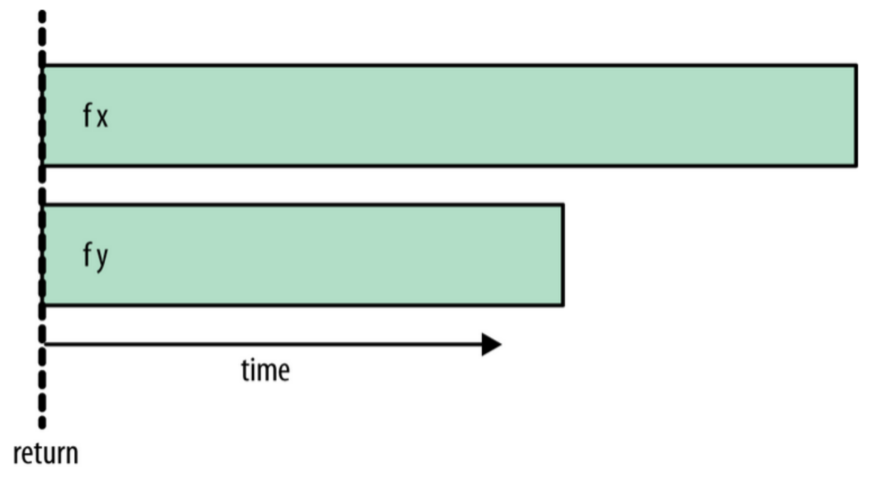


Figure 2-5. rpar/rpar timeline

```
type  Strategy a = a -> Eval a

parPair :: Strategy (a,b)
parPair (a,b) = do
                      a' <- rpar a
                      b' <- rpar b
                      return (a',b')
main = print pair
          where
            pair = runEval ( parPair (fib 35, fib 36))
```

Strategiile sunt impementate in modulul [Control.Parallel.Strategies]

Pentru a ilustra posibilitatile de implementare si extindere a functionalitatilor vom crea operatii noi folosind din modulul Control.Parallel.Strategies numai rpar

```
type MyStrategy a = a -> Eval a
```

```
myrparforce :: NFData a => MyStrategy a
myrparforce x = rpar (force x)
```

o strategie pentru evaluare in paralel, argumentul fiind complet evaluat

Strategii parametrizate

```
myevalPair :: MyStrategy a -> MyStrategy b -> MyStrategy (a,b)
myevalPair st1 st2 (x,y) = do
                              x' <- st1 x
                              y' <- st2 y
                              return (x',y')
```

st1 este argument de tip **MyStrategy a**
st2 este argument de tip **MyStrategy b**

```
myparPair :: (NFData a, NFData b) => MyStrategy (a,b)
myparPair = myevalPair myrparforce myrparforce
```

st1 = myrparforce a
st2 = myrparforce b

```
main = print pair
          where
             pair = runEval ( myparPair (fib 36, fib 35))
```

Strategii parametrizate

```
myparList :: NFData a => MyStrategy a -> MyStrategy [a]
myparList strat [] = return []
myparList strat (x:xs) = do
                        x' <- strat x
                        xs'<- myparList strat xs
                        return (x':xs')
```

strat este un parametru de tip **MyStrategy a**

myparList este o strategie de evaluare a listelor, parametrizata de strategia de evaluare a elementelor