Laziness and Infinite Datastructures

Koen Lindström Claessen

A Function

```
fun :: Maybe Int -> Int
fun mx | mx == Nothing = 0
       | otherwise     = x + 3
 where
  x = fromJust mx
```

Could fail… What happens?

Another Function

```
dyrt :: Integer -> Integer
dyrt n | n <= 1    = 1
       | otherwise = dyrt (n-1) + dyrt (n-2)
```

```
choice :: Bool -> a -> a -> a
choice False x y = x
choice True  x y = y
```

```
Main> choice False 17 (dyrt 99)
17
```

Without delay…

Laziness

- Haskell is a *lazy* language
  - Things are evaluated *at most once*

  - Things are only evaluated when they are needed
  - Things are never evaluated twice

Understanding Laziness

- Use error or undefined to see whether something is evaluated or not
  - choice False 17 undefined
  - head [3,undefined,17]
  - head (3:4:undefined)
  - head [undefined,17,13]
  - head undefined

Lazy Programming Style

- Separate
  - Where the computation of a value is defined
  - Where the computation of a value happens

Modularity!

Lazy Programming Style

- head [1..1000000]
- zip ”abc” [1..9999]
- take 10 [’a’..’z’]
- …

When is a Value ”Needed”?

```
strange :: Bool -> Integer
strange False = 17
strange True  = 17
```

```
Main> strange undefined
Program error: undefined
```

An argument is evaluated when a *pattern match* occurs

But also *primitive functions* evaluate their arguments

And?

```
(&&) :: Bool -> Bool -> Bool
True  && True  = True
False && True  = False
True  && False = False
False && False = False
```

What goes wrong here?

And and Or

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && x = False
```

```
(||) :: Bool -> Bool -> Bool
True  || x = True
False || x = x
```

```
Main> 1+1 == 3 && dvrt 99 == dvrt 99
False
```

```
Main> 2*2 == 4 || undefined
True
```

At Most Once?

```
apa :: Integer -> Integer
apa x = f x + f x
```

f x is evaluated twice

```
bepa :: Integer -> Integer -> Integer
bepa x y = f 17 + x + y

Main> bepa 1 2 + bepa 3 4
310
```

Quiz: How to avoid recomputation?

f 17 is evaluated twice

At Most Once!

```
apa :: Integer -> Integer
apa x = fx + fx
 where
  fx = f x
```

```
bepa :: Integer -> Integer -> Integer
bepa x y = f17 + x + y

f17 :: Integer
f17 = f 17
```

Example: BouncingBalls

```
type Ball = [Point]

bounce :: Point -> Int -> Ball
bounce (x,y) v
  | v == 0 && y == maxY = []
  | y' > maxY           = bounce (x,y) (2-v)
  | otherwise           = (x,y) : bounce (x,y') (v+1)
 where
  y' = y+v
```

Generates a long list…

But when is it evaluated?

Example: Sudoku

```
solve :: Sudoku -> Maybe Sudoku
solve sud
  | ...
  | otherwise =
       listToMaybe
       [ sol
       | n <- [1..9]
       , ... solve (update p (Just n) sud) ...
       ]
```

”Generate and test”

Infinite Lists

- Because of laziness, values in Haskell can be *infinite*
- Do not compute them completely!
- Instead, only use parts of them

Examples

- Uses of infinite lists
  - `take n [3..]`
  - ``xs `zip` [1..]``

Example: PrintTable

```haskell
printTable :: [String] -> IO ()
printTable xs =
  sequence_ [ putStrLn (show i ++ ":" ++ x)
            | (x,i) <- xs `zip` [1..]
            ]
```

lengths adapt
to each other

Iterate

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

```
Main> iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512,1024,...
```

Other Handy Functions

```
repeat :: a -> [a]
repeat x = x : repeat x

cycle :: [a] -> [a]
cycle xs = xs ++ cycle xs
```

Quiz: How to define these with iterate?

Alternative Definitions

```
repeat :: a -> [a]
repeat x = iterate id x

cycle :: [a] -> [a]
cycle xs = concat (repeat xs)
```

Problem: Replicate

```
replicate :: Int -> a -> [a]
replicate = ?

Main> replicate 5 ’a’
”aaaaa”
```

Problem: Replicate

```
replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)
```

Problem: Grouping List Elements

```
group :: Int -> [a] -> [[a]]
group = ?

Main> group 3 ”apabepacepa!”
[”apa”,”bep”,”ace”,”pa!”]
```

Problem: Grouping List Elements

```
group :: Int -> [a] -> [[a]]
group n = takeWhile (not . null)
        . map (take n)
        . iterate (drop n)
```

. connects
”stages” --- like
Unix pipe symbol **|**

Problem: Prime Numbers

```
primes :: [Integer]
primes = ?

Main> take 4 primes
[2,3,5,7]
```

Problem: Prime Numbers

```
primes :: [Integer]
primes = 2 : [ x | x <- [3,5..], isPrime x ]
 where
  isPrime x =
    all (not . (`divides` x))
      (takeWhile (\y -> y*y <= x) primes)
```

Infinite Datastructures

```
data Labyrinth
  = Crossroad
  { what  :: String
  , left  :: Labyrinth
  , right :: Labyrinth
  }
```

How to make an interesting labyrinth?

Infinite Datastructures

```
labyrinth :: Labyrinth
labyrinth = start
 where
  start  = Crossroad ”start”  forest town
  town   = Crossroad ”town”   start  forest
  forest = Crossroad ”forest” town   exit
  exit   = Crossroad ”exit”   exit   exit
```

What happens when we print this structure?

Laziness: Summing Up

- Laziness
  - Evaluated at most once
  - Programming style
- Do not have to use it
  - But powerful tool!
- Can make programs more ”modular”

(primes race)

Side-Effects

- Writing to a file
- Reading from a file
- Creating a window
- Waiting for the user to click a button
- ...
- Changing the value of a variable

Pure functions cannot / should not do this

That's why we use *instructions* (a.k.a. monads)

Benefit?

Pure Computations

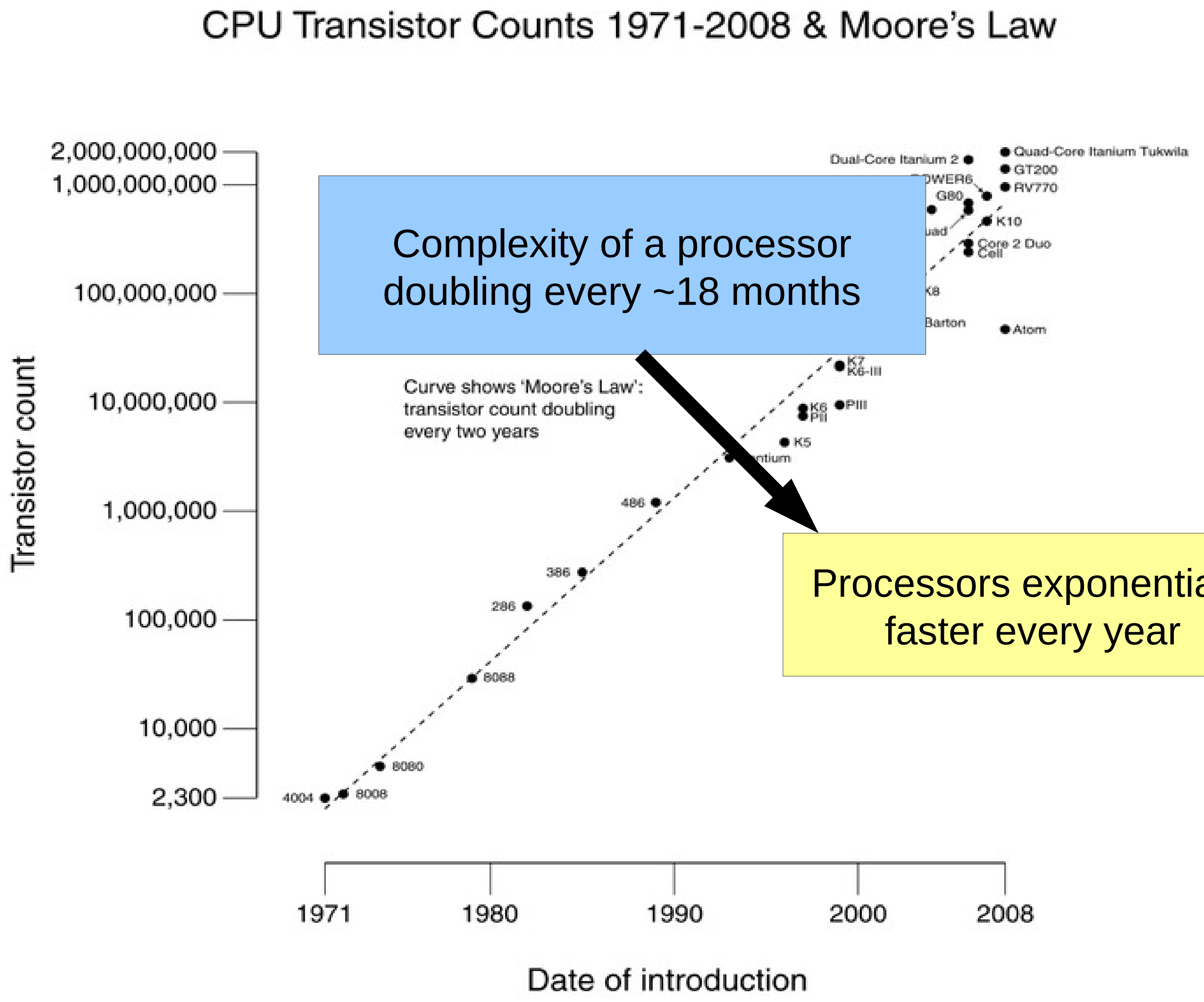
- Can be evaluated whenever
  - no side effects
  - the same result
- If no-one is interested in the result
  - do not compute the result!

- Pure functions are *required* for laziness

“Imperative Programming”

- Imperative programming
  - side effects are the main reason to run a computation

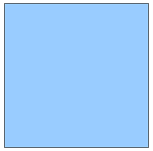
- Functional programming
  - computation results are the main (only) reason to run a computation



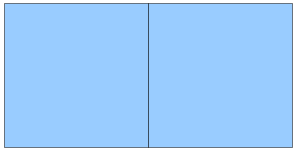
Complexity of a processor doubling every ~18 months

Processors exponentially faster every year

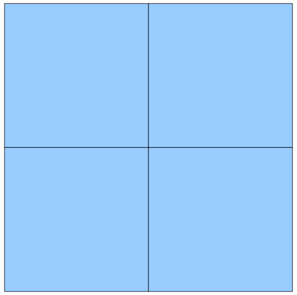
Processors Today and Tomorrow



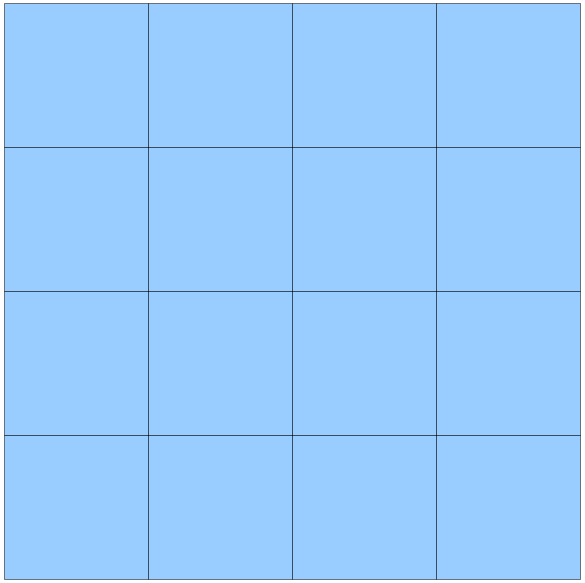
single
core



dual core



quadcore



16 core

32 core

64 core

128 core

How to program these?

Parallelism

- Previously, computation went one step at a time
- Now, we can (and have to) do *many things at the same time, “in parallel”*

- Side effects and parallelism do not mix well: *race conditions*

Parallelism in Haskell

- **import** Control.Parallel

```
seq :: a -> b -> b
```

seq x y:
“first evaluate x, then
produce y as a result”

```
par :: a -> b -> b
```

par x y:
“produce y as a result,
but also evaluate x
in parallel”

Parallelism in Haskell

```
parList :: [a] -> b -> b
parList []     y = y
parList (x:xs) y = x `par` (xs `parList` y)
```

```
pmap :: (a->b) -> [a] -> [b]
pmap f xs = ys `parList` ys
 where
  ys = map f xs
```

(understand the result:
remove all the pars)

Parallelism in Haskell (2)

```
data Expr = Num Int
          | Add Expr Expr
```

```
peval :: Expr -> Int
peval (Num n)   = n
peval (Add a b) = x `par` y `par` x+y
 where
  x = peval a
  y = peval b
```

Parallelism in Haskell (3)

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = left ++ [x] ++ right
 where
  left  = qsort [ y | y <- xs, y <= x ]
  right = qsort [ y | y <- xs, y > x ]
```

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = left  `parList`
               right `parList`
               left ++ [x] ++ right
 where
  left  = qsort [ y | y <- xs, y <= x ]
  right = qsort [ y | y <- xs, y > x ]
```

Pure Functions...

- ...enable easier *understanding*
  - only the arguments affect the result
- ...enable easier *testing*
  - stimulate a function by providing arguments
- ...enable *laziness*
  - powerful programming tool
- ...enable easy *parallelism*
  - no head-aches because of side effects

(understand the result:
remove all the pars)

Do’s and Don’ts

Repetitive code – hard to see what it does...

```
lista :: a -> [a]
lista x = [x,x,x,x,x,x,x,x,x]
```

```
lista :: a -> [a]
lista x = replicate 9 x
```

Do’s and Don’ts

```
siffra :: Integer -> String
siffra 1 = ″1″
siffra 2 = ″2″
siffra 3 = ″3″
siffra 4 = ″4″
siffra 5 = ″5″
siffra 7 = ″7″
siffra 8 = ″8″
siffra 9 = ″9″
siffra _ = ″###″
```

Repetitive code – hard to see what it does...

Is this really what we want?

```
siffra :: Integer -> String
siffra x | 1 <= x && x <= 9 = show x
         | otherwise        = ″###″
```

Do’s and Don’ts

How much time does this take?

```
findIndices :: [Integer] -> [Integer]
findIndices xs = [ i | i <- [0..n], (xs !! i) > 0 ]
 where
  n = length xs-1
```

```
findIndices :: [Integer] -> [Integer]
findIndices xs = [ i | (x,i) <- xs `zip` [0..], x > 0 ]
```