# Introduction to Cloud Computing

## Functional Programming and MapReduce

15-319, spring 2010

13th Lecture, March 9th

**Iliano Cervesato**

# Lecture Goals

- **Introduction to functional programming**
- **Understand how MapReduce was designed by borrowing elements from functional programming and deploy them in a distributed setting**
- **Introduction to MapReduce program model**
  - Advantages and why it makes sense

# Lecture Outline

- **Functional programming**
  - Introduction
  - Map
  - Fold
  - Examples
  - Exploiting parallelism in map
- **MapReduce**

# Functional Programming

- **Not to be confused with imperative / procedural programming**
  - Think of mathematical functions and λ Calculus
  - Computation is treated as evaluation of expressions and functions on lists containing data
  - Apply functions on data to transform them

# Functional Programming Characteristics

- **Data structures are persistent**
  - Functional operations do not modify data structures
    - New data structures are created when an operation is performed
    - Original data still exists in unmodified form
  - Data flows are implicit in the program design
  - No state
- **Functions are treated as first-class entities in FP**
  - Can be passed to and returned by functions
  - Can be constructed dynamically during run-time
  - Can be a part of data structures

# A Simple Example - Factorial

- Consider the factorial in mathematics
- Mathematical definition

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n((n-1)!) & \text{if } n > 0 \end{cases} \qquad \forall n \in \mathbb{N}.$$

# C Program to Evaluate Factorial

- An Iterative program to evaluate factorial
- We describe the "steps" needed to obtain the result
- But is it really equivalent to factorial?

```
int factorial (int n) {
        f =0;
        while(n>0) {
                f = f*n;
                n--;
        }
        return f;
}
```

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n((n-1)!) & \text{if } n > 0 \end{cases} \quad \forall n \in \mathbb{N}.$$

- Observation: The program changes the state of variables f and n during execution
- You describe the steps necessary to perform the computation, going to the level of the machine

# Factorial Function in ML

- In Standard ML

```
fun factorial (n:int): int =
    if n = 0
        then 1
    else n * factorial(n-1)
```

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n((n-1)!) & \text{if } n > 0 \end{cases} \qquad \forall n \in \mathbb{N}.$$

- Function definition mirrors the mathematical definition
- No concept of state, n does not get modified
- Functional programming allows you to describe computation at the level of the problem, not at the level of the machine

# A Functional Programming Example in C

- **Functional programming is not an attribute of the language but a state of mind**
  - We can rewrite the factorial program recursively in C as follows:

```
int factorial (int n)
{
    if (n == 0) return 1;
    else
          return n * factorial (n-1);
}
```

- **C does support some aspects of functional programming but emphasizes imperative programming**

# Examples of Functional Languages

- **Lots of examples:**
  - LISP – One of the oldest, but outdated
  - Scheme
  - ML, CAML etc.
  - JavaScript, Python, Ruby
- **Functional programming compilers/interpreters have to convert high level constructs to low-level binary instructions**
- **Myth: Functional programming languages are inefficient**
  - By and large a thing of the past,
  - Modern compilers generate code that is close to imperative programming languages

# Lists in Functional Programming

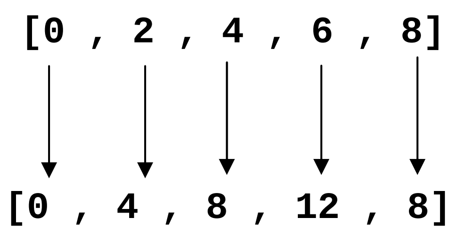
- A List is a collection of elements in FP (usually of the same type)
- Example:
  - `val L1 = [0,2,4,6,8]`
    `val L2 = 0::[2,4,6,8]`
  - `::` (cons) is the constructor operator in ML, **`nil`** represents the empty list

# Operations on Lists - I

- **Let's define a double operation on a list as follows:**

```
fun double nil = 0
   |double [x::L] = 2 * x :: double L
```

- **This function can be computed as follows:**




This is a common type of operation in FP and can be expressed as a **map operation**

- **Many functions work this way and can be expressed also as a map operation**
- **These functions operate on each list element independently.**
  - They can be parallelized

# The Map Operation

- A Map function is used to apply an operation to every element of a list

  - ```
    fun map nil = nil
          | map f(x::L) = (f x) :: map of L
    ```

  - ```
    fun twice x = 2 * x
    ```

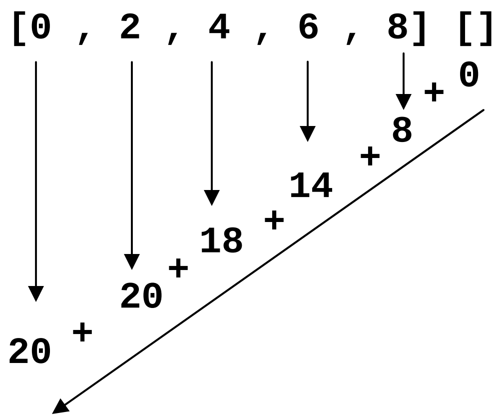
  - ```
    fun double L = map twice L
    ```
    ```

# Operations on Lists - II

- **Let's define a sum operation on a list as follows:**

```
fun sum nil = 0
   |sum [x::L] = x + sum L
```

- **This function can be computed as follows:**

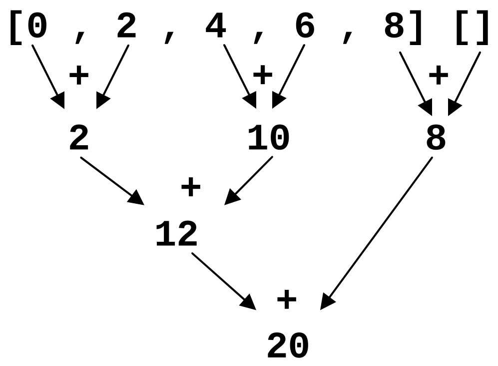



This is a common type of operation in FP and can be expressed as a **fold operation**

- **The computation happens from left to right and takes n steps**
  - But since the sum operation is associative, it doesn't have to be so. This does not work for non-associate functions (such as subtract)

# Parallelism in List Operations

- If an operation is associative, if can be evaluated as follows:




- Here the operation is done in O(log n) time.

# The Fold Operation

- **Fold operation is used to combine elements of a list**
  - Two functions: **`foldl`** and **`foldr`** for ‘fold left’ and ‘fold right’
  - For associative functions, they produce the same result.

```
fun foldr f b nil = b
  | foldr f b (x::l) = f(x, foldr f b l)
```

  - This function is equivalent to:

```
foldr f b [x1,x2,...,xn] = f(x1, f(x2, ..., f(xn,b)...)
```

# Implicit Parallelism in List Functions

- **In a purely functional setting, calls to f on each element of a list are independent**
  - Can be parallelized.
- **If order of application of *f* to elements in list is *associative*, we can reorder or parallelize execution of *f***
- **This is the "secret" that MapReduce exploits**