F# Overview: Immutable Data + Pure Functions

Acknowledgements

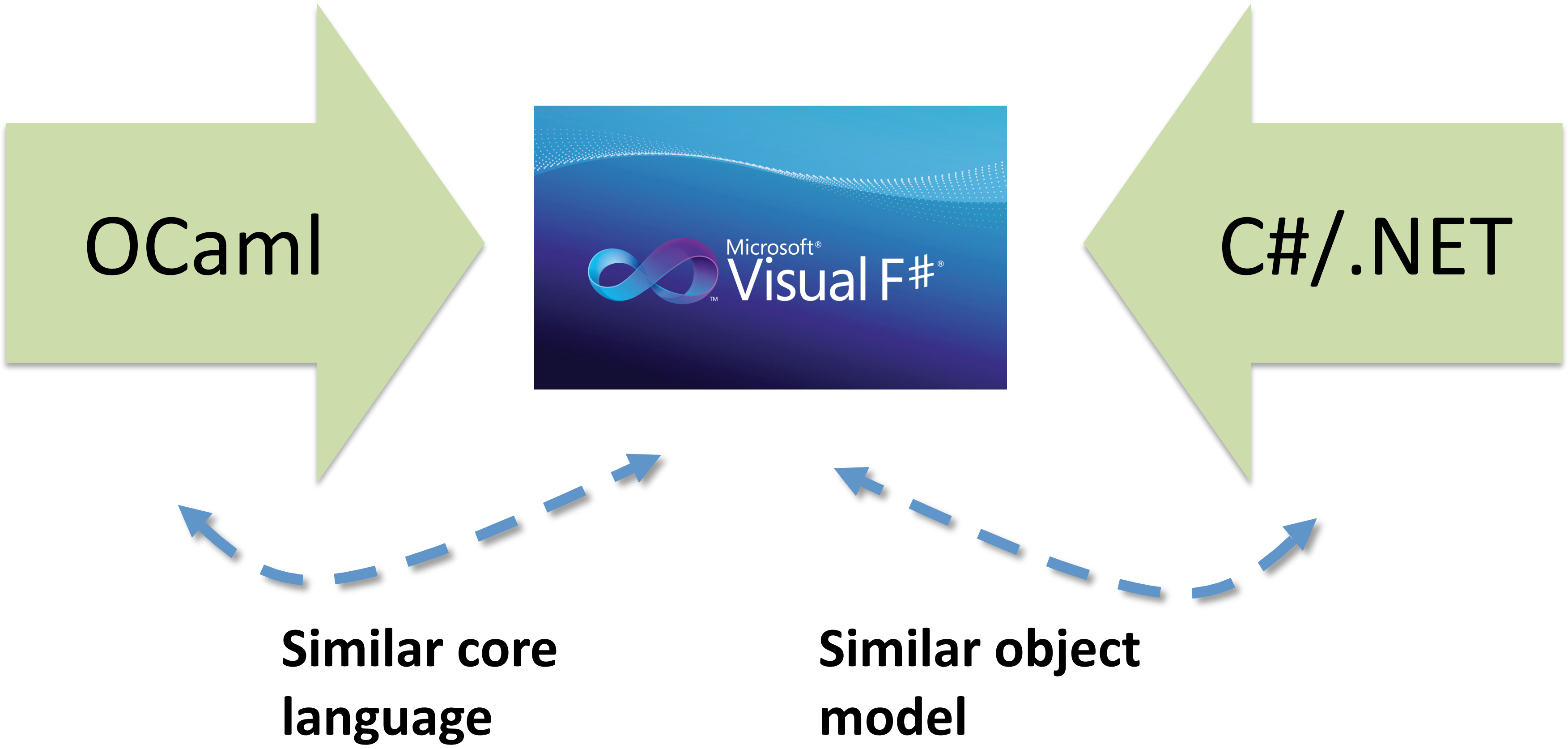
- Authored by
  - Thomas Ball, MSR Redmond

- Includes content from the F# team

Functional Languages

- Focus on data
  - Immutability
  - Applicative data transformations
    - map, filter, reduce, …
- Pure functions
  - can execute in parallel without interference

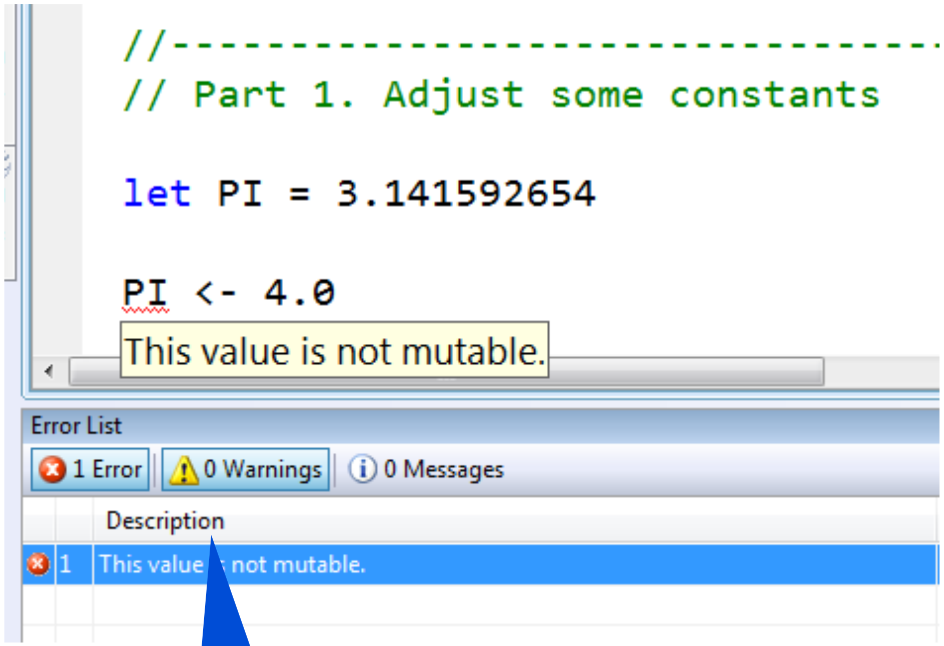
F#: Influences



Immutability is the Default in F#

- **Immutable Lists!**
- **Immutable Records!**
- **Immutable Sets!**
- **Immutable Objects!**
- **Immutable Tuples!**
- **Immutable Dictionaries!**
- **Immutable Unions!**
- **+ lots of language features to encourage immutability**

Immutability the norm…



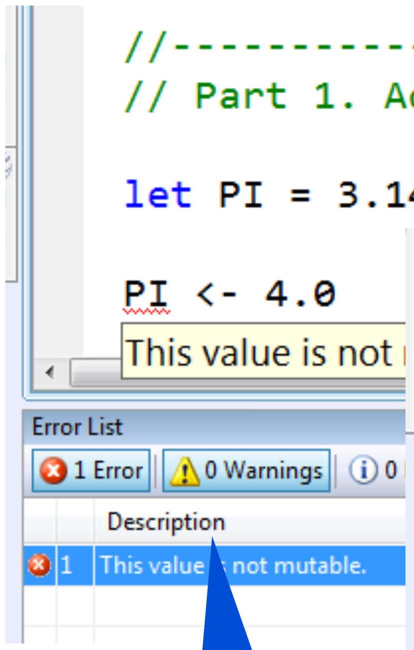
```
//--------------------------------
// Part 1. Adjust some constants

let PI = 3.141592654

PI <- 4.0
```

This value is not mutable.

1 This value is not mutable.

Values may not be changed

Immutability the norm…



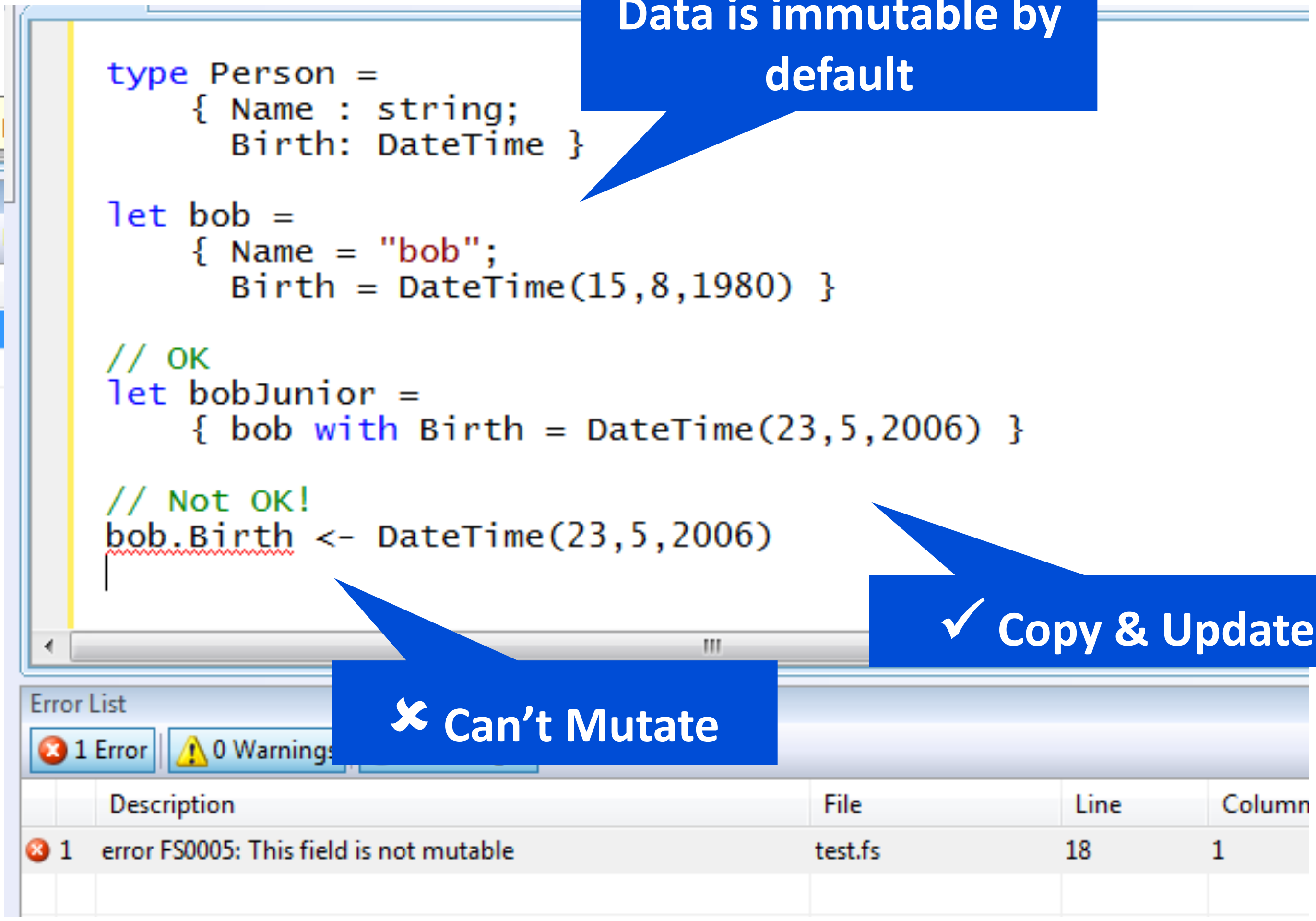
```
//-----------------------------------
// Part 1. Adjust some constants

let PI = 3.141592654

PI <- 4.0
```

This value is not

Error List

1 Error | 0 Warnings | 0

		Description
✕	1	This value is not mutable.



```
type Person =
    { Name : string;
      Birth: DateTime }

let bob =
    { Name = "bob";
      Birth = DateTime(15,8,1980) }

// OK
let bobJunior =
    { bob with Birth = DateTime(23,5,2006) }

// Not OK!
bob.Birth <- DateTime(23,5,2006)
```

Error List

1 Error | 0 Warnings

		Description	File	Line	Column
✕	1	error FS0005: This field is not mutable	test.fs	18	1

Data is immutable by default

Values may not be changed

✓ Copy & Update

✘ Can't Mutate

In Praise of Immutability

Immutable objects can transfer between threads

Immutable objects never have race conditions

Some Basic Types

Basic Types/Literals	
`int`	`76`
`string`	`"abc", @"c:\etc"`
`float`	`3.14, 3.2e5`
`char`	`'7'`
`bool`	`true, false`
`unit`	`()`

Some Basic Operators

Overloaded Arithmetic

x + y	**Addition**
x - y	**Subtraction**
x * y	**Multiplication**
x / y	**Division**
x % y	**Remainder/modulus**
-x	**Unary negation**

Booleans

not *expr*	**Boolean negation**
expr* && *expr	**Boolean “and”**
expr* || *expr	**Boolean “or”**

Tuples

Tuple	*Type*
`(1,2)`	`int*int`
`(1,2,3)`	`int*int*int`
`(1,2,3,4)`	`int*int*int*int`
`((1,2),3)`	`(int*int)*int`
`(true,(2,3))`	`bool*(int*int)`

Let Bindings (give names to values)

Let "let" simplify your life…

Type inference. The **safety** of C# with the **succinctness** of a scripting language

Bind a static value

Bind a static function

Bind a local value

Bind a local function

```
let data = (1, 2, 3)

let f (a, b, c) =
    let sum = a + b + c
    let g x = sum + x*x
    (g a, g b, g c)
```

Lists

`[]`	Empty list
`[x]`	One element list
`[x;y]`	Two element list
`hd::tl`	Cons element hd on list tl
`l1@l2`	Append list l2 to list l1

`length l`	number of elements in l
`map f l`	map function f over l
`filter f l`	elements of l passing f
`zip l1 l2`	One list from two lists

Recursive Polymorphic Data Types

```
type 'a Tree =
  | Leaf of 'a
  | Node of 'a Tree list

let tree0 = Leaf (1,2)
let tree1 = Node [Leaf (2,3);Leaf (3,4)]
let tree2 = Node [t0;t1]
```

Lambdas in F#

```
let timesTwo = List.map (fun i -> i*2) [1;2;3;4]
```

```
> let timesTwo = List.map (fun i -> i*2) [1;2;3;4];;

val timesTwo : int list = [2; 4; 6; 8]
```

```
let sumPairs = List.map (fun (a,b) -> a+b) [(1,9);(2,8);(3,7)]
```

```
> let sumPairs = List.map (fun (a,b) -> a+b) [(1,9);(2,8);(3,7)];;

val sumPairs : int list = [10; 10; 10]
```

Function Application

```
> let data = (1, 2, 3) ;;

val data : int * int * int = (1, 2, 3)

> let f (a, b, c) =
    let sum = a + b + c
    let g x = sum + x*x
    (g a, g b, g c) ;;

val f : int * int * int -> int * int * int

> let res = f data ;;

val res : int * int * int = (7, 10, 15)
```

Function Currying

```
> List.map ;;

val it : (('a -> 'b) -> 'a list -> 'b list)

> let timesTwoFun = List.map (fun i -> i*2) ;;

val timesTwoFun : (int list -> int list)

> timesTwoFun [1;2;3] ;;

val it : int list = [2; 4; 6]
```

Functional– Pipelines

The pipeline operator

```
x |> f
```

```
f x
```

Functional– Pipelines

Successive stages in a pipeline

```
x |> f1
  |> f2
  |> f3
```

```
f3 (f2 (f1 x))
```

Pattern Matching

```
match expr with
| pat -> expr
…
| pat -> expr
```

Matching Basic Values

```
/// Truth table for AND via pattern matching

let testAndExplicit x y =
    match x, y with
    | true, true -> true
    | true, false -> false
    | false, true -> false
    | false, false -> false
```

Truth table

```
> testAndExplicit true true;;

true
```

Wildcards

```
/// Truth table for AND via pattern matching

let testAnd x y =
    match x, y with
    | true, true -> true
    | _ -> false
```

“Match anything”

```
> testAnd true false;;

false
```

Matching Structured Data

```
let rec length l =
    match l with
    | []          -> 0
    | _::tl       -> 1+(length tl)
```

A series of structured patterns

```
> listLength [1;2;3] ;;

3
```

Two Popular List Functions

```
let rec map f l =
    match l with
    | []       -> []
    | hd::tl   ->
        (f hd)::(map f tl)
```

('a -> 'b) -> 'a list -> 'b list

```
let rec fold f acc l =
    match l with
    | [] -> acc
    | hd::tl ->
       (fold f (f acc hd) tl)
```

('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

"Aggregation"

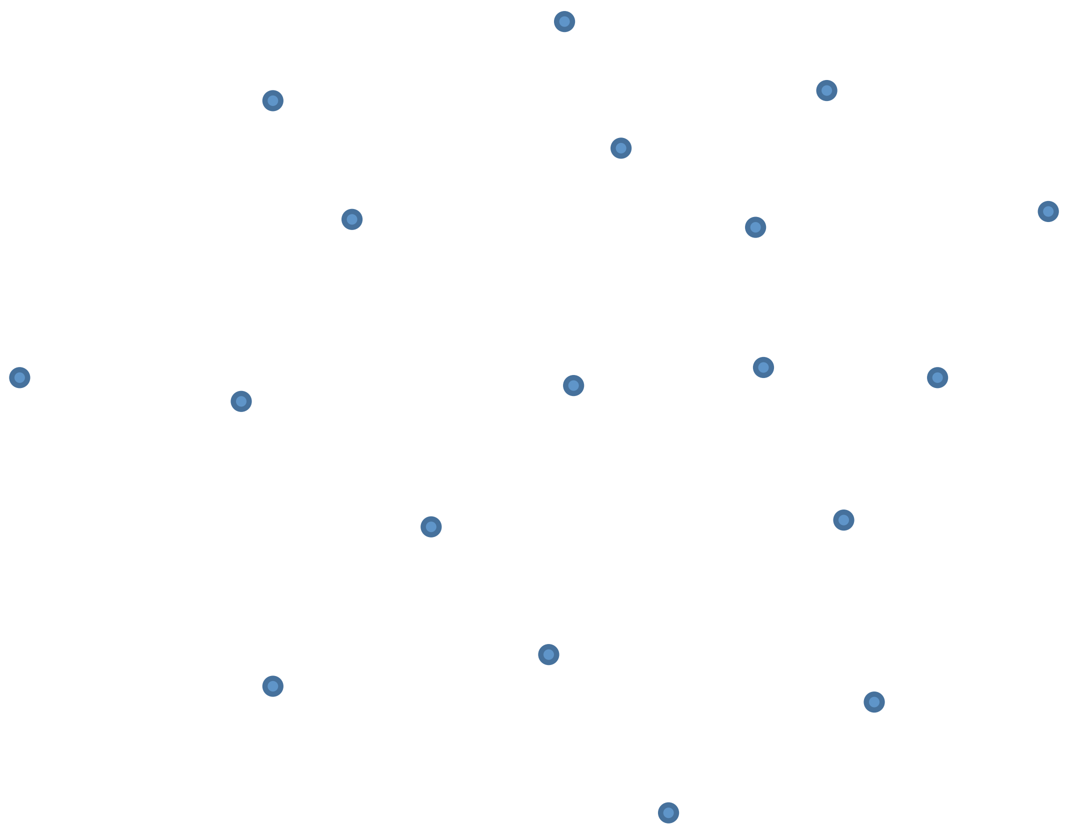
Matching On a Tree

```
let tree0 = Leaf (1,2)
let tree1 = Node [Leaf (2,3);Leaf (3,4)]
let tree2 = Node [t0;t1]
```

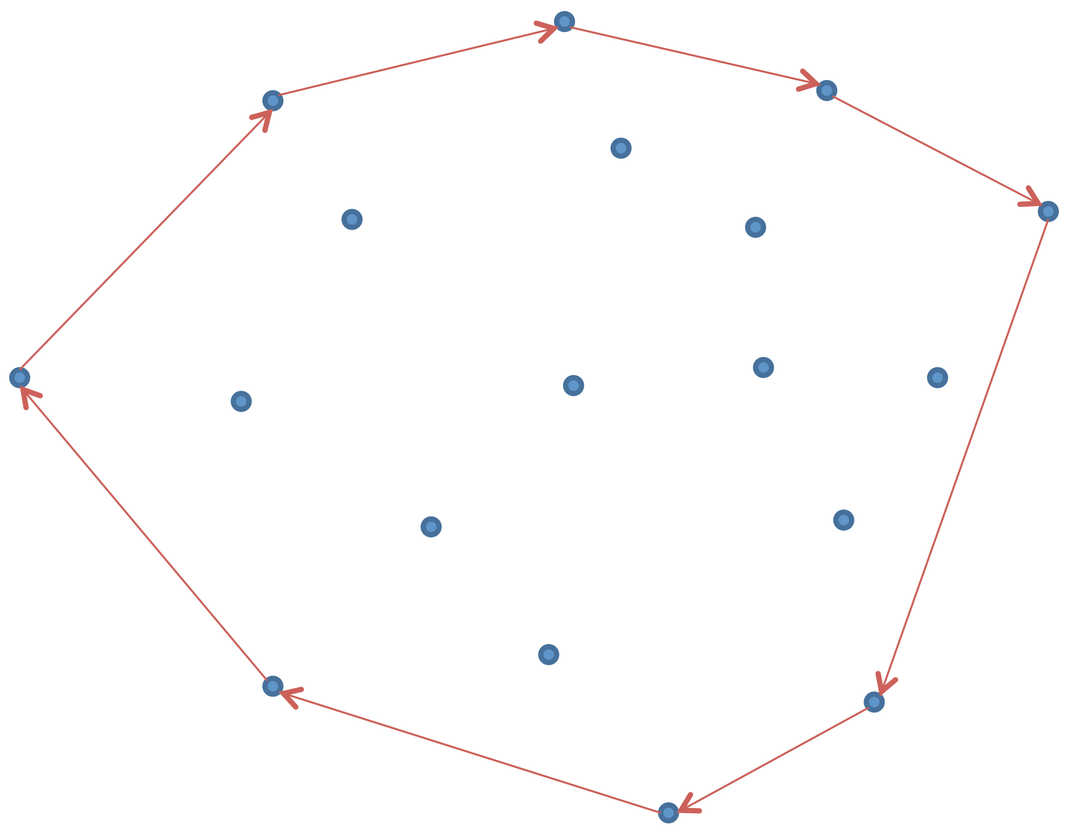
```
let rec size t =
    match t with
    | Leaf _     -> 1
    | Node l     -> fold (fun a t -> a+(size t)) 1 l
```

```
> size tree2 ;;

5
```

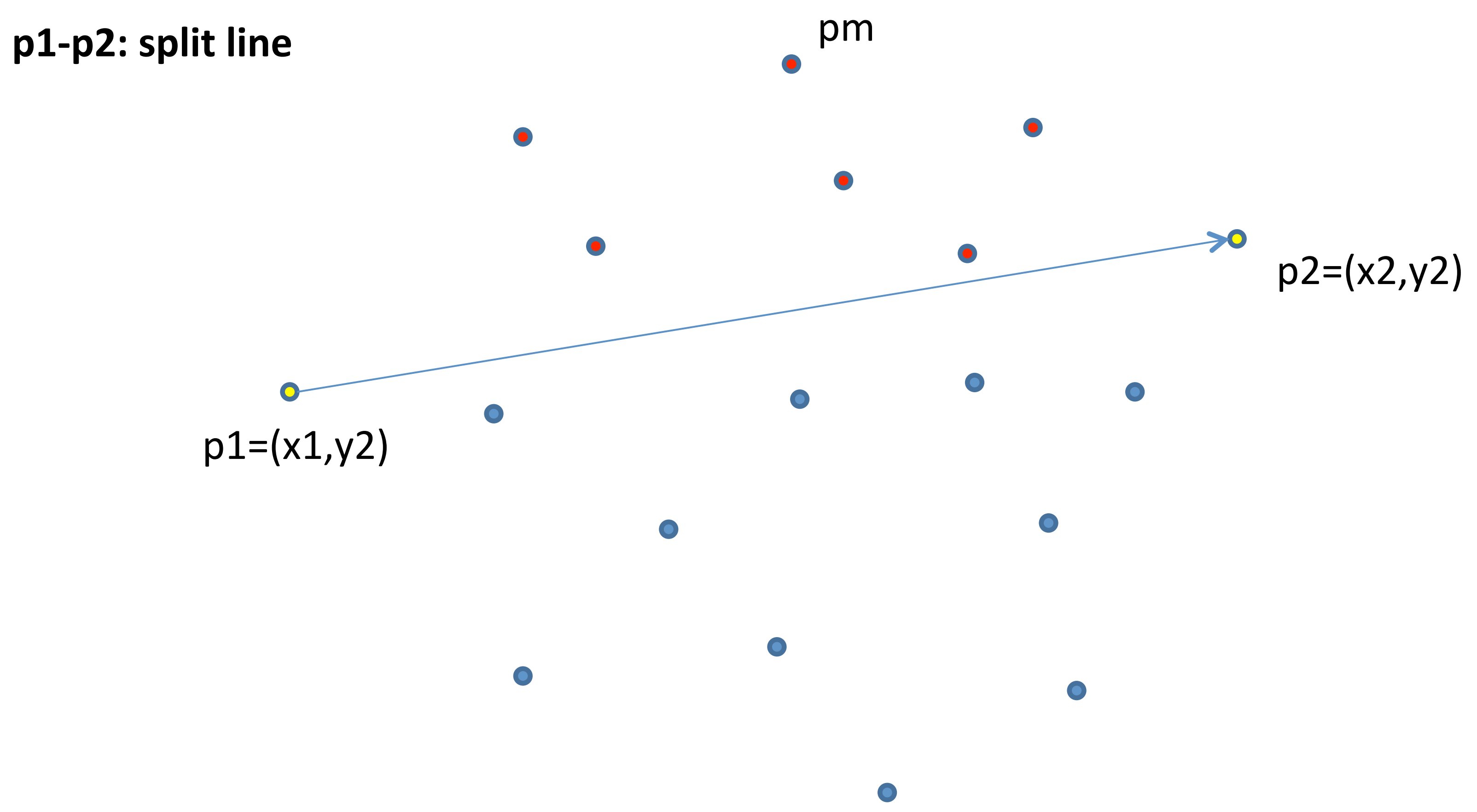
Application: Compute Convex Hull



Application: Compute Convex Hull

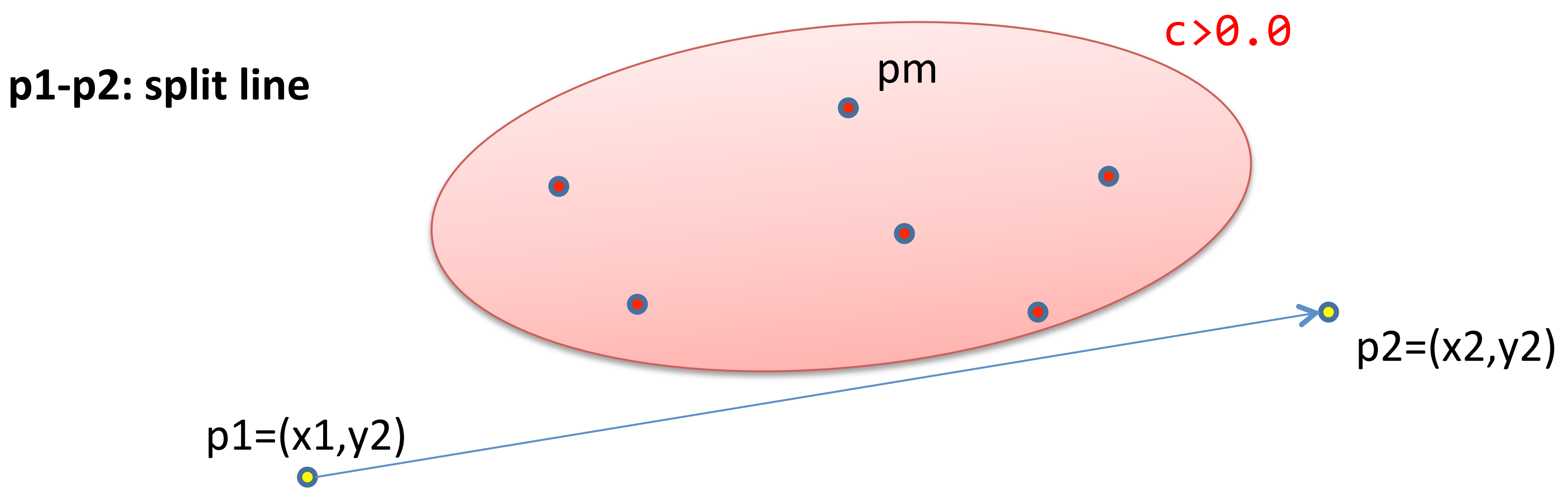


QuickHull, Pictorially



```
let cross_product ((x1,y1),(x2,y2)) (xo,yo) = (x1-xo)*(y2-yo) - (y1-yo)*(x2-xo)
```

Points Above Line with Distance



```
let accFun line ((pm,max),l) p =
    let cp = cross_product line p
    if (cp > 0.0) then
        ((if cp > max then (p,cp) else ((pm,max))), p::l)
    else
        ((pm,max),l)

let aboveLineAndMax points line =
    points
    |> List.fold accFun (((0.0,0.0),0.0),[])
```

QuickHull

```
let rec hsplit points (p1,p2) =
    let ((pm,_),aboveLine) = aboveLineAndMax points (p1,p2)
    match aboveLine with
    | [] | _::[] ->
        p1::aboveLine
        |> HullLeaf
    | _ ->
        [(p1,pm);(pm,p2)]
        |> List.map (hsplit aboveLine)
        |> HullNode

let quickhull points =
    let minx = List.minBy (fun (x,_) -> x) points
    let maxx = List.maxBy (fun (x,_) -> x) points
    [(minx,maxx);(maxx,minx)]
    |> List.map (hsplit points)
    |> HullNode
```

Next Lecture

- Parallelizing QuickHull

Parallelizing QuickHull

- Most of the computation takes place in `aboveLineAndMax`

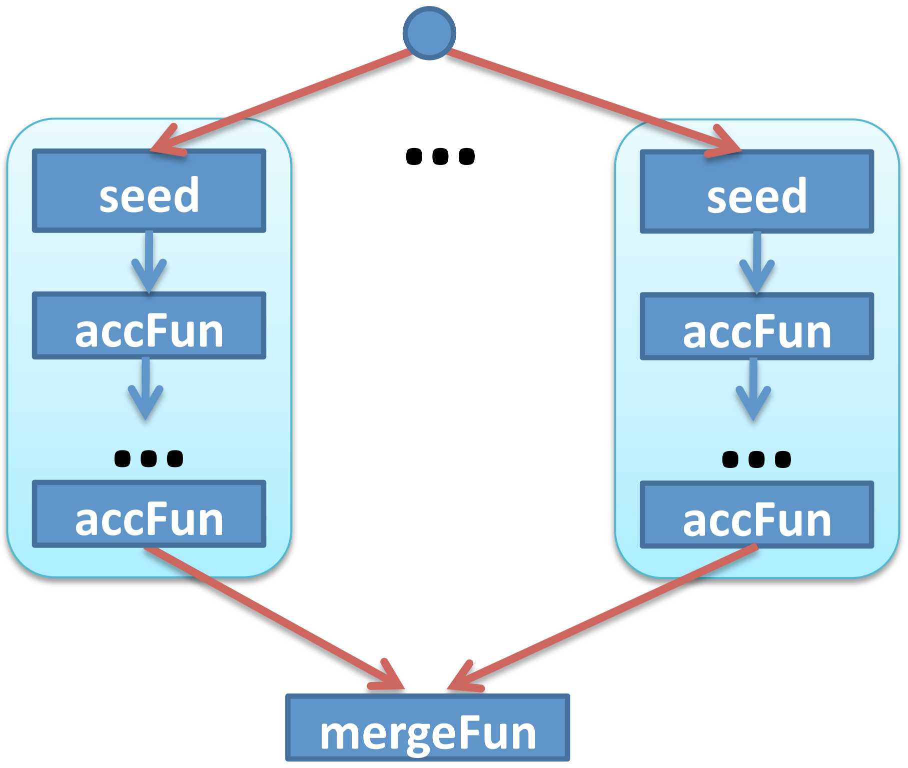
```
let aboveLineAndMax points line =
    points
    |> List.fold accFun (((0.0,0.0),0.0),[])
```

- `List.fold accFun seed ls`
  - `accFun: 'Acc -> 'S -> 'Acc`
  - `seed: 'Acc`
  - `ls: 'S list`

Sequential Implementation of Fold

```
let rec fold accFun seed ls =
    match ls with
    | [] -> seed
    | hd::tl ->
       (fold accFun (accFun seed hd) tl)
```

Parallel Aggregation Pattern



Parallel Aggregation Implementation (using .NET Tasks)

```
let rec foldPar accFun mergeFun seed listOfLists =
    let accL s l = l |> List.fold accFun s |> finalFun
    listOfLists
    |> List.map (fun ls -> Task.Factory.StartNew(accL seed ls))
    |> List.map (fun task -> task.Result)
    |> List.fold mergeFun (finalFun seed)
```

This is the useful Map/Reduce pattern

Correctness?

- Parallel aggregation partitions the input and uses the same **seed** multiple times
- Parallel aggregation does not necessarily apply **accFun** in the same order as the sequential aggregation
- Parallel aggregation uses **mergeFun** to merge results from different partitions

Associativity/Commutativity of Operator **F**

- For all valid inputs x, y, z:
  - **F**(x,y) is *associative* if **F(F(x,y),z) = F(x,F(y,z))**
  - **F**(x,y) is *commutative* if **F(x,y) = F(y,x)**

- For example, Max is commutative because
  - Max(x,y) = Max(y,x)
- And also associative because
  - Max(Max(x,y),z) = Max(x,Max(y,z))

Associativity/Commutativity Examples

		Associative	
		No	Yes
Commutative	No	(a, b) => a / b (a, b) => a – b (a, b) => 2 * a + b	(string a, string b) => a.Concat(b) (a, b) => a (a, b) => b
	Yes	(float a, float b) => a + b (float a, float b) => a * b (bool a, bool b) => !(a && b) (int a, int b) => 2 + a * b (int a, int b) => (a + b) / 2	(int a, int b) => a + b (int a, int b) => a * b (a, b) => Min(a, b) (a, b) => Max(a, b)

Three Correctness Rules

- Let **S**=seed, **F**=accL, **G**=mergeFun

1. F(a, x) = G(a, F(S, x))

– for all possible accumulator values of **a** and all possible element values of **x**

2. G(a, b) = G(b, a)

– for all possible values of **a**, **b**

3. G(G(a, b), c) = G(a, G(b, c))

– for all possible values of **a**, **b**, **c**

Something To Prove

- Given
  - list L = `l1@...@lN`
  - **seed**, **accFun**, **mergeFun** obeying three rules

- Show

```
List.fold accFun seed l1@...@lN |> finalFun
                        =
   foldPar accFun mergeFun seed [l1;…;lN]
```

Performance

```
let rec foldPar accFun finalFun mergeFun seed listOfLists =
    let accL s l = l |> List.fold accFun s
    listOfLists
    |> List.map (fun l -> Task.Factory.StartNew(accL seed l))
    |> List.map (fun task -> task.Result)
    |> List.fold mergeFun (finalFun seed)
```

- Concerns
  - accL should do enough computation to offset cost of coordination (fork/join of Tasks)
  - sublists of listOfLists should be of sufficient size and balanced

Returning to QuickHull

```
let accFun line ((pm,max),(len,l)) p =
    let cp = cross_product line p
    if (cp > 0.0) then
        ((if cp > max then (p,cp) else ((pm,max))), (len+1,p::l))
    else
        ((pm,max),(len,l))

let aboveLineAndMax points line =
    points
    |> List.fold accFun (((0.0,0.0),0.0),(0,[]))
```

finalFun and **mergeFun** for QuickHull?

```
let mergeFun ((pm1,max1),(cnt1,l1)) ((pm2,max2),(cnt2,l2)) =
    ((if (max1 > max2) then (pm1,max1) else (pm2,max2)),
     (cnt1+cnt2,l1@l2))
```

Problem: l1@l2 expensive for large lists

Correctness

- l1@l2 is associative but not commutative
- Doesn't matter because we are treating list of lists as a set of lists (so we are using @ as a union operator)

QuickHull Issues

- The size of the point sets can shrink considerably for each recursive call to hsplit
- Track size of output of `aboveLineAndMax` to determine
  - if parallelization of futures calls will be worthwhile
  - when to return to fully sequential processing

Where does the List of Lists (Partition) Come From?

- Static partition of initial list
- Dynamic partition