CSCI 334:
Principles of Programming Languages

Lecture 12-2: ML and F#

Instructor: Dan Barowy

Williams

Outline

- Algebraic data types (ADTs)
- ADTs and pattern matching: better together
- Using patterns to avoid errors
- Higher-order functions: map and fold

Algebraic Data Types*

*not to be confused with Abstract Data Types!

Algebraic Data Type

An **algebraic data type** is a composite data type, made by combining other types in one of two different ways:

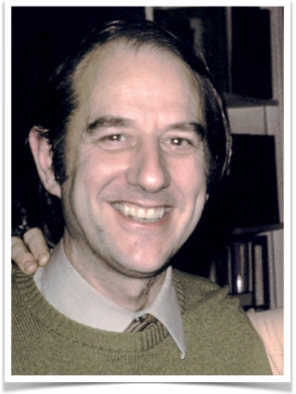
- by **product**, or
- by **sum**.

You've already seen **product types**: tuples and records.

So-called b/c the set of all possible values of such a type is the cartesian product of its component types.

We'll focus on **sum types**.

Algebraic Data Types



- Invented by Rod Burstall at University of Edinburgh in '70s.
- Part of the HOPE programming language.
- Not useful without pattern matching.
- Like peanut butter and chocolate, they are "better together."

A "move" function in a game

A "move" function in a game (Java)

```
public static final int NORTH = 1;
public static final int SOUTH = 2;
public static final int EAST = 3;
public static final int WEST = 4;

public … move(int x, int y, int dir) {
  switch (dir) {
    case NORTH: ...
    case ...
  }
}
```

A "move" function in a game (Java)

Discriminated Union (sum type)

```
type Direction =
     North | South | East | West;

let move coords dir =
  match coords,dir with
  |(x,y),North -> (x,y - 1)
  |(x,y),South -> (x,y + 1)
```

- Above is an "incomplete pattern"
- ML will warn you when you've missed a case!
  - "proof by exhaustion"

Parameters

```
type Shape =
    | Rectangle of float * float
    | Circle of float
```

- Pattern match to extract parameters

```
let s = Rectangle(1.0,4.0)
match s with
| Rectangle(w,h) -> …
| Circle(r) -> …
```

Named parameters

```
type Shape =
    | Rectangle of width: float * height: float
    | Circle of radius: float
```

- Names are really only useful for initialization, though.

```
let s = Rectangle(height = 1.0, width = 4.0)
```

ADTs can be recursive and generic

```
type MyList<'a> =
      | Empty
      | NonEmpty of head: 'a * tail: MyList<'a>
```

```
> NonEmpty(2, Empty);;
    val it : MyList<int> = NonEmpty (2,Empty)
```

Avoiding errors with patterns

- Another example: handling errors.
- SML has exceptions (like Java)
- But an alternative, easy way to handle many errors is to use the option type:

```
type option<'a> =
| None
| Some of 'a
```

Avoiding errors with patterns

```
let divide quot div =
  match div with
  | 0 -> None
  | _ -> Some (float quot/float div)
```

Avoiding errors with patterns

```
> divide 6 7;;
val it : float option = Some 0.8571428571

> divide 6 0;;
val it : float option = None

>
```

option type

- Why `option`?
- `option` is a data type;

  not handling errors is a static type error!

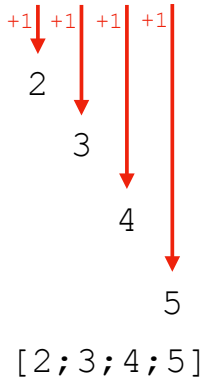
Mapping and Folding

map

Intuition:

map

```
List.map (fun x -> x + 1) [1;2;3;4];
```




```
[2;3;4;5]
```

map

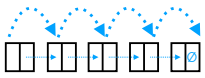
```
[2;8;22;4]
|> List.map (fun x -> x + 1)
|> List.map float
|> List.map (fun x -> x / 3.3)
|> List.sort
```

```
[0.9090909091; 1.515151515; 2.727272727;
6.96969697]
```

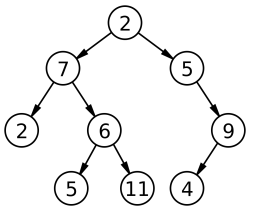
fold

structural recursion → fold it!

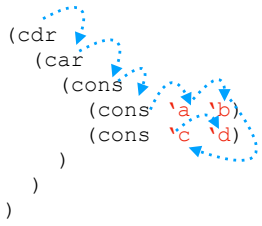
(in a nutshell: any problem that recurses on a subset of input)



list length



tree height



evaluation

fold

Intuition:

fold left

```
List.fold (fun acc x -> acc+x) 0 [1;2;3;4]
```

```
acc = 0, [1;2;3;4]
acc = 0+1, [2;3;4]
acc = 1+2, [3;4]
acc = 3+3, [4]
acc 6+4, []
returns acc = 10
```

fold right

```
List.foldBack
  (fun x acc -> acc+x) [1;2;3;4] 0
```

```
[1;2;3;4], acc = 0
[1;2;3], acc = 0+4
[1;2], acc = 4+3
[1] acc = 7+2
[], acc = 9+1
returns acc = 10
```

fold

- If you haven't done the collaborative activity yet, STOP.
- Write a function `number_in_month` that takes a list of dates (where a date is `int*int*int`) and an `int month` and returns how many dates are in `month`
- Use fold

fold

```
let number_in_month(ds: Date list)(month: int) : int =
  ds
  |> List.fold (fun acc (_,mm,_) ->
       if month = mm then
         acc + 1
       else
         acc
     ) 0
```

Recap & Next Class

Today we covered:

- ADTs
- Pattern matching with ADTs
- Avoiding errors with option types
- Map and fold

Next class:

- Parser combinators