Intersection and union types

Dr. Giuseppe Maggiore

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Lecture topics

- Implicitly building inclusive data-structures with tuples
- Explicitly building inclusive data-structures with records
- Implicitly building exclusive data-structures with `Choice`
- Explicitly building exclusive data-structures with discriminated unions
- Pattern-matching

```
let i : int = 10
let f : float = 3.0
let g : float32 = 3.0f
let b : bool = true
let s : string = "Hi! I am a string. Hurr durr."
let point : float * float = (3.0, 4.0)

let twoStrings : string * string = s, "Another string
    !?!?!"
```

```
let pointAndStrings : (float * float) * (string *
    string) = point, twoStrings
```

```
type Point = float * float
```

```
let get_x (p:Point) =
  let (x,y) = p
  x

let get_y (p:Point) =
  let (x,y) = p
  y
```

```
let (+) (p1:Point) (p2:Point) =
  let (x1,y1) = p1
  let (x2,y2) = p2
  x1+x2,y1+y2
```

```
let rec travel (p:Point) (v:Point) =
  do printfn "%A" p
  let _ = System.Console.ReadLine()
  travel (p+v) v
travel (0.0,0.0) (1.0,0.0)
```

Introduction

Records

- Sometimes tuples are not expressive enough
- `type Ship = Point * Point * float * float * float`

Introduction

Records

- Sometimes tuples are not expressive enough
- `type Ship = Point * Point * float * float * float`
- With *records* we can give names to fields

```
type Point = { X : float; Y : float }
```

```
let (+) (p1:Point) (p2:Point) =
  { X = p1.X + p2.X
    Y = p1.Y + p2.Y }
```

```
let rec travel (p:Point) (v:Point) =
  do printfn "%A" p
  let _ = System.Console.ReadLine()
  travel (p+v) v
travel { X = 0.0; Y = 0.0 } { X = 0.0; Y = 0.0 }
```

Introduction

Units of measure

- Type constraints on records (and any other types, but less used)
- Units of measure: restrict composition on values of the same type

```
type [<Measure>] m
type [<Measure>] s
type Point<[< Measure>] ’a>  =
  { X : float<’a>; Y : float<’a> }
```

```
let (*) (p:Point<’a>) (k:float<’b>) =
  { X = p.X * k
    Y = p.Y * k }

let (+) (p1:Point<’a>) (p2:Point<’a>) =
  { X = p1.X + p2.X
    Y = p1.Y + p2.Y }
```

```
let rec travel (p:Point<m>) (v:Point<m/s>) (dt:float<s
    >) =
  do printfn "%A" p
  let _ = System.Console.ReadLine()
  travel (p + v * dt) v dt
travel { X = 0.0<m>; Y = 0.0<m> }
       { X = 1.0<m/s>; Y = 0.0<m/s> }
       0.1<s>
```

```
type Point<[<Measure>] 'a>  = { X : float<'a>; Y :
    float<'a> }
  with
    static member (*) (p:Point<'a>, k:float<'b>) =
      { X = p.X * k
        Y = p.Y * k }
    static member (+) (p1:Point<'a>, p2:Point<'a>) =
      { X = p1.X + p2.X
        Y = p1.Y + p2.Y }
```

Introduction

Discriminated unions

- Tuples and records are many shapes joined into one
- Sometimes a value may take one out of multiple possible shapes
- We use *discriminated unions in this case*

```
type IntOrError =
  | Int of int
  | Error of string
```

```
let addPositive (x:IntOrError) (y:int) =
  match x with
  | Int i -> Int(i + y)
  | Error(e) -> Error(e)
```

```
let addPositive (x:IntOrError) (y:int) =
  match x with
  | Int i ->
    let res = i + y
    if res < 0 then
      Error "Not positive!"
    else
      Int(res)
  | Error(e) -> Error(e)
```

```
type ValueOrError<’T> =
  | Value of ’T
  | Error of string
```

```
let addPositive (x:ValueOrError<int>) (y:int) =
  match x with
  | Value i ->
    let res = i + y
    if res < 0 then
      Error "Not positive!"
    else
      Value(res)
  | Error(e) -> Error(e)
```

```
type Option<’T> =
  | Some of ’T
  | None
```

```
type WarpStatus =
  | Charging of float<s>
  | Charged
```

```
type SpaceShip = {
    Position   : Point<m>
    Velocity   : Point<m/s>
    WarpEngine : WarpStatus
  }
```

```
let rec travel (s:SpaceShip) (dt:float<s>) =
  do printfn "%A" s
  let _ = System.Console.ReadLine()
  match s.WarpEngine with
  | Charging(timeLeft) when timeLeft > 0.0<s> ->
    let s' =
      { s with
          Position   = s.Position + s.Velocity * dt
          WarpEngine = Charging(timeLeft - dt) }
    travel s' dt
  | Charging(timeLeft) ->
    let s' =
      { s with
          Position   = s.Position + s.Velocity * dt
          WarpEngine = Charged }
    travel s' dt
  | Charged ->
    let s' =
      { s with
          Position   = s.Position + s.Velocity * dt *
              100.0
          WarpEngine = Charging(10.0<s>) }
```

```
do travel { Position   = { X = 0.0<m>; Y = 0.0<m> }
            Velocity   = { X = 1.0<m/s>; Y = 0.0<m/s>
                }
            WarpEngine = Charged }
          2.0<s>
```

Introduction

Conclusions and assignment

- The assignments are on Natschool
- **Restore** the games to a working state
- Hand-in a **printed** report that only contains your **sources** and the associated **documentation**

Introduction

Conclusions and assignment

- Any book on the topic will do
- I did write my own (Friendly F#) that I will be loosely following for the course, **but it is absolutely not mandatory or necessary to pass the course**

Dit is het

The best of luck, and thanks for the attention!