Records and abstract data types

Dr. Giuseppe Maggiore

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Lecture topics

- Abstract data types and interfaces as records of functions
- Abstract operations as requiring records of functions to work

```
let add x y = x + y
let a = add 1.0 2.0
let b = add 1 2
let c = add 1.0f 2.0f
let d = add "one" "two"
let add (+) x y = x + y
let a = add (+) 1.0 2.0
let b = add (+) 1 2
let c = add (+) 1.0f 2.0f
let d = add (+) "one" "two"
let line (+) (*) a b x =
  a * x + b
```

```
let b = line (+) (*) 1.0 2.0 3.0
let line (+) (*) a b x =
  a * x + b
```

```
let a = line (+) (*) 1 2 3
let b = line (+) (*) 1.0 2.0 3.0
```

Introduction

Grouping operations together

- Instead of the single operations, we could give a record of them
- This record is an abstraction over all the data types that will be able to support its operations

```
type NumberOperations<’a> =
  { plus  : ’a -> ’a -> ’a
    times : ’a -> ’a -> ’a }
```

```
let line ops a b x =
  let (+) = ops.plus
  let (*) = ops.times
  a * x + b
```

```
let intOps   : NumberOperations<int>   = { plus = (+);
    times = (*) }
let floatOps : NumberOperations<float> = { plus = (+);
    times = (*) }
```

```
let a = line intOps 1 2 3
let b = line floatOps 1.0 2.0 3.0
```

Introduction

Creating a library of abstract operations

- `plus` and `times` really look the same
- We could build a record of operations for both of them
- `Number` is then two of these records of operations

```
type Combine<'a> = { Empty : 'a; Append : 'a -> 'a ->
    'a }
  with static member Create z (+) = { Empty = z;
      Append = (+) }
```

```
let intPlus    = Combine<int>.Create 0 (+)
let intTimes   = Combine<int>.Create 1 (*)
let floatPlus  = Combine<float>.Create 0.0 (+)
let floatTimes = Combine<float>.Create 1.0 (*)
let stringPlus = Combine<string>.Create "" (+)
let listPlus() = Combine<List<'a>>.Create [] (@)
let setPlus()  = Combine<Set<'a>>.Create Set.empty (+)
let sumStuff ops a b c =
  let (+) = ops.Append
  a + b + c + c
```

```
sumStuff intPlus 10 20 30
sumStuff floatPlus 10.0 20.0 30.0
sumStuff stringPlus "a" "b" "c"
sumStuff (listPlus()) [1] [2] [3;4]
sumStuff (listPlus()) ["1"] ["2"] ["3";"4"]
```

```
type Number<'a> =
  { Plus  : Combine<'a>
    Times : Combine<'a> }
  with static member Create p t = { Plus = p; Times =
     t }
```

```
let line ops a b x =
  let (+) = ops.Plus.Append
  let (*) = ops.Times.Append
  a * x + b
```

```
let intOps     = Number<int>.Create intPlus intTimes
let floatOps   = Number<float>.Create floatPlus
    floatTimes
```

```
let a = line intOps 1 2 3
let b = line floatOps 1.0 2.0 3.0
```

Introduction

Creating an automated hierarchy of abstract operations

- We can build generic combinators that transform our records of functions
- This allows us to **automatically** extend our library

```
let optionCombine (c:Combine<’a>) : Combine<Option<’a
   >> =
  Combine<Option<’a>>.Create (Some c.Empty) (fun (x) (
     y) -> match x, y with Some x, Some y -> Some(c.
     Append x y) | _ -> None)
```

```
let optionNumber (n:Number<'a>) : Number<Option<'a>> =
  Number<Option<'a>>.Create (optionCombine n.Plus) (
      optionCombine n.Times)
```

```
line (intOps |> optionNumber) (Some 3) (Some 10) (Some
     5)
```

```
let pairCombine (c1:Combine<’a>) (c2:Combine<’b>) :
    Combine<’a * ’b> =
  Combine<’a * ’b>.Create (c1.Empty, c2.Empty) (fun (
      x1,y1) (x2,y2) -> c1.Append x1 x2, c2.Append y1
      y2)
```

```
let pairNumber (c1:Number<'a>) (c2:Number<'b>) :
    Number<'a * 'b> =
  Number<'a * 'b>.Create (pairCombine c1.Plus c2.Plus)
       (pairCombine c1.Times c2.Times)
```

```
line (pairNumber intOps floatOps) (3, 1.0) (4, 2.0)
    (5, 6.0)
```

```
line (pairNumber intOps floatOps |> optionNumber) (
    Some(3, 1.0)) (Some(4, 2.0)) (Some(5, 6.0))
```

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**

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!