F# Core

Рустам Шехмаметьев

03.11.2018

SAR
DOT
NET

Structure

- F# overview
- Values
- Types and type inference
- Functions
- Pattern matching
- Lists and list comprehension

Brief overview

- Multiparadigm (leans to functional programming)
- Cross-platform (thanks to .NET Core)
- Interoperable with other .NET languages

Hello, World

F#

```
open System

[<EntryPoint>]
let main argv =
    printfn "Hello World!"
    0
```

C#

```
using System;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Functional programming

- Program is an evaluation of mathematical (pure) functions
- Avoids changing state and mutable data
- Requires ‘impure’ functions to work

Values

Values

- Declared with ‘let’
- Functions are values too!

```
let message = "Nice to meet you"
let sayTo name message = printf "Hi %s. %s" name message
let anotherSayTo = sayTo
```

Values

- Immutable by default
- Can be made mutable with ‘mutable’ keyword

```
let mutable mutMessage = message
message <- sayTo “Mark“ “How is your sex life?" // Compilation error
mutMessage <- "Is this a Room reference" // Ok
```

Why immutability?

```
var ten = 10;
DoWork(ten);
var result = ten + 1;
// ??
```

```
function DoWork(arg) {
  ten = "Hello, world"
}
```

```
class Message
{
    public string Body { get; set; }
}
```

```
var message = new Message()
{ Body = "Hello, world" };
var newMessage = TransformMessage(message);
// message.Body??
```

Types

Types

- Declared with ‘type’ keyword
- Non-nullable by default
- Interoperable with .NET types

.NET types

```
type Student(id: Guid, name: string, surname: string) =
    new(name, surname) = Student(Guid.NewGuid(), name, surname)
    member this.Name = name
    member this.Surname = surname
    member this.Id = id
    member self.FullName = self.Name + " " + self.Surname
```

Algebraic types

- Divided into 2 categories: “AND” and “OR”
- Don’t have any methods

Algebraic types

- Have ToString by default

```
type User = { Id: int; Username: String }

let vasya = { Id = 1; Username = "Vasya" }

// { Id = 1; Username = "Vasya" }
printfn "%s" (vasya.ToString())
```

Algebraic types

- Have Equals by default
- Use structural equality

```
let vasya = { Id = 1; Username = "Vasya" }

let vasya2 = { Id = 1; Username = "Vasya" }

printfn "%b" (vasya = vasya2) // true
```

AND types (tuples)

```
// int * string
let user = (1, "user")

printfn "%i" (fst user) // 1

printfn "%s" (snd user) // “user”
```

AND types (records)

```
type User = { Id: int
              Username: string}

let user = { Id = 1; Username = "User" }

let username = user.Username

let incompleteUser = { Id = 3 } // Compilation error
```

Generics

```
type KeyValuePair<'a, 'b> = {
    Key: 'a
    Value: 'b
}

let intStringPair = { Key = 1; Value = "Hello" }
let intIntPair = { Key = 1; Value = 1 }

type AddressBook = {
    Addresses: string List // List<string>
}
```

OR types

F#

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

let someTen = Some 10

let someHello = Some "Hello"

let nothing = None
```

C#

```
public abstract class Option<T>
{
}

public class Some<T> : Option<T>
{
    public T Value { get; }

    public Some(T value) => Value = value;
}

public class None<T> : Option<T>
{
}
```

OR types

```
type UserId = UserId of int

type WorkWeek = Monday | Tuesday | Wednesday | Thursday | Friday

type WorkWeekEnum = Monday = 1 | Tuesday = 2 ...
```

Function types

- Denoted by arrows
- Last type in the arrow chain is the return type

```
// int -> int -> int
let add a b = a + b
```

Function types

- “unit” type means absence of value (void)
- Created with “()”

```
// type: unit -> unit
let sayHello() = printf "Hello“


// type: string -> unit
let sayHelloTo s = printf "Hello, %s" s
```

Function types

```
// type: (int -> int) -> (int -> int) -> int
// returns (number + 2) * 3
let add2mul3 add2 mul3 number = mul3 (add2 number)
```

Type inference

- Types are deduced by the compiler

```
// type: (int -> string) -> int -> string
let someFunc f x =
    let y = f (x + 1)
    "Value is " + y
```

Type inference

- Types can be explicitly annotated

```
let someFunc (f: int -> string)
             (x: int) : string = ...
```

Type inference

- If type cannot be inferred then it’s generalized

```
// type: 'a -> 'a
let identity x = x

// type: int
let ten = identity 10

// type: string
let hello = identity "Hello"
```

Type inference (Algebraic types)

```
type User = {
    Id: int
    Username: string
}

let getUsername user = user.Username
```

Type inference (.NET types)

```
type User(id: int, name:string) =
    member x.Id = id
    member x.Username = name

let getUsername user = user.Username // Compilation Error
let getUsername (user:User) = user.Username // Ok
```

Type inference

F#

```
let join outer
         inner
         outerKeySelector
         innerKeySelector
         resultSelector
```

C#

```
IEnumerable<TResult>
Join<TOuter, TInner, TKey,
TResult>(this
IEnumerable<TOuter> outer,
IEnumerable<TInner> inner,
Func<TOuter, TKey>
outerKeySelector,
Func<TInner, TKey>
innerKeySelector,
Func<TOuter, TInner, TResult>
resultSelector)
```

Functions

Lambdas

- Lambdas are created using “fun” keyword
- Multiple parameters are separated with spaces

```
let add = fun x y -> x + y
```

Lambdas

- Lambdas and functions are the same

```
// int -> int -> int
let add a b = a + b

// int -> int -> int
let add = fun x y -> x + y
```

Operators

- Operators are functions
- Name should only have symbols

```
// int -> int -> int
let (^) x n = pown x n

// 16
let resultPow = 2 ^ 4
```

Functional composition

- Functions are building blocks of an app
- Compose them together to create new functions!
- You can compose functions by using “>>” operator
- Composition only works on functions with 1 parameter

```
// ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)
let (>>) f1 f2 = fun x ->  f2 (f1 x)
```

Functional composition example

- Task: calculate user's purchasing power
- Subtasks (for one user):
  - Get user info from DB
  - Calculate purchasing power based on that info
  - Write changes to DB

Functional composition example

```
// int -> User
let getUserFromDb id = ...

// User -> User
let calculateBuyingPower user = ...

// User -> unit
let writeUserToDb user = ...

// int -> unit
let processUser = getUserFromDb >> updateUserDateFormat >> writeUserToDb
```

Pipes

```fsharp
// 'a -> ('a -> 'b) -> b
let (|>) x f = f x

let add2 a = a + 2

let mul3 a = a * 3

let result = 10 |> add2 |> add2 |> mul3 // 42

let awfulResult = mul3 (add2 (add2 (10))) // 42
```

Currying

- Allows to represent functions with many parameters as a function with 1 parameter

```
// int -> int -> int -> int
1. let add a b c = a + b + c

// int -> int -> int -> int
2. let add = fun x y z -> x + y + z

// int -> int -> int -> int
3. let add = fun x -> (fun y z -> x + y + z)

// int -> int -> int -> int
4. let add = fun x -> (fun y -> (fun z -> x + y + z))
```

Partial application

- Allows to create a new function from an existing one
- Can be used for dependency injection

```
// int -> int -> int
let add a b = a + b

// int -> int
let add2 = add 2
```

But what about C#?

```
public static Func<T1, T3> Compose<T1, T2, T3>(this Func<T1,
T2> func1, Func<T2, T3> func2) => x => func2(func1(x));

public static T2 Pipe<T1, T2>(this T1 value,
                              Func<T1, T2> func) =>
func(value);

public static Func<T1, Func<T2, T3>> Curry<T1, T2, T3>(this
Func<T1, T2, T3> func) => x => y => func(x, y);
```

Recursion

- Functions need to have “rec” modifier to be recursive
- Tail call optimization makes recursion a valid alternative to loops

```
let rec factorial n =
     if n <= 0
     then 1
     else n * factorial(n - 1)
```

Recursion and cycles

```
for x in [1..10] do
    printfn "%i" x
```

```
[1..10]
|> List.iter (printfn "%i")
```

Pattern matching

F#

```
let someTen = Some 10

let message =
    match someTen with
    | None -> "None"
    | Some value -> "Some"
```

C#

```
Option<int> someTen = new Some<int>(10);
string message;

switch (someTen)
{
    case None<int>:
        message = "None";
        break;
    case Some<int> value:
        message = "Some";
        break;
}
```

Pattern matching

```
let user = { Id = 1
             Username = "user" }

match user with
| { Id = 1; Username = username } -> ...
| { Id = id; Username = "user" } -> ...
| _ -> ...
```

Destructuring

```fsharp
let (a, b, c) = (1, 2, 3)

// ('a * 'b * 'c) -> 'c
let thrd (a, b, c) = c
thrd (1, 2, 3) |> printf "%i" // 3

// User -> string
let getUsername ({ Id = _; Username = username }) = username

// Option<'a> -> 'a
// Dangerous. Can cause runtime error
let getSomeValue (Some value) = value

getSomeValue None // Runtime error
```

Lists

Lists

- Implemented as a linked list
- It is immutable (operations return new list)

Lists

- Initialized with []
- Append with “::” operator
- Concatenate with “@” operator

```
let list = [1;2;3]

let newList = 4 :: list // [4;1;2;3]

let concatenatedLists = list @ newList //[1;2;3;4;1;2;3]
```

List

- Functions to work with lists are in the “List” module

```
let list = [1..10]
           |> List.filter (fun x -> x % 2 = 0)
           |> List.sum
```

List comprehension

- Can comprehend lists with simple comprehension [start..step..stop]

```
let oneToTen = [1..10]

let fiveToOneStepTwo = [5..-2..1] // [5;3;1]

let alphabet = ['A'..'Z']
```

List comprehension

- Can comprehend with complex for .. yield

```
// ['a'..'z']
let alphabetLower = [for x in alphabet do yield Char.ToLower x]
```

```
let evenPairs = [for i in [1..10] do
                     for j in [1..10] do
                         if (i + j) % 2 = 0 then yield (i, j)]
```

The Good…

- Functional by default
- Combines the best of OOP and functional programming
- Expressive and concise

…and the Bad

- Can be less effective than C#
- Steep learning curve
- Not many projects

Resources

- MSDN: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference
- F# for fun and profit: https://fsharpforfunandprofit.com/

Rustam Shekhmametyev

shekhmametyev.rustam@gmail.com