Типы и генерики в F#

Юрий Литвинов

12

Шаблонные типы

```
type 'a list = ...
type list<'a> = ...

List.map : ('a -> 'b) -> 'a list -> 'b list
let map<'a,'b> : ('a -> 'b) -> 'a list -> 'b list =
    List.map

let rec map (f : 'a -> 'b) (l : 'a list) =
    match l with
    | h :: t -> (f h) :: (map f t)
    | [] -> []
```

Автоматическое обобщение

```
let getFirst (a, b, c) = a
let mapPair f g (x, y) = (f x, g y)
```

F# Interactive

```
val getFirst: 'a * 'b * 'c -> 'a
val mapPair : ('a -> 'b) -> ('c -> 'd)
    -> ('a * 'c) -> ('b * 'd)
```

Алгоритм Евклида, не генерик

```
let rec hcf a b =
    if a = 0 then b
    elif a < b then hcf a (b - a)
    else hcf (a - b) b
```

F# Interactive

```
val hcf : int -> int -> int

> hcf 18 12;;
val it : int = 6

> hcf 33 24;;
val it : int = 3
```

Алгоритм Евклида, генерик

```
let hcfGeneric (zero, sub, lessThan) =
    let rec hcf a b =
        if a = zero then b
        elif lessThan a b then hcf a (sub b a)
        else hcf (sub a b) b
    hcf

let hcfInt = hcfGeneric (0, (-), (<))
let hcfInt64 = hcfGeneric (0L, (-), (<))
let hcfBigInt = hcfGeneric (0I, (-), (<))
```

F# Interactive

```
val hcfGeneric: 'a * ('a -> 'a -> 'a) * ('a -> 'a -> bool)
    -> ('a -> 'a -> 'a)
```

Словари операций

```
type Numeric<'a> =
    { Zero: 'a;
      Subtract: ('a -> 'a -> 'a);
      LessThan: ('a -> 'a -> bool); }

let hcfGeneric (ops : Numeric<'a>) =
    let rec hcf a b =
        if a = ops.Zero then b
        elif ops.LessThan a b then hcf a
            (ops.Subtract b a)
        else hcf (ops.Subtract a b) b
    hcf
```

Примеры использования

```
let intOps = { Zero = 0;
    Subtract = (-);
    LessThan = (<) }

let bigintOps = { Zero = 0I;
    Subtract = (-);
    LessThan = (<) }

let hcfInt = hcfGeneric intOps
let hcfBigInt = hcfGeneric bigintOps
```

Повышающий каст

F# Interactive

```
> let xobj = (1 :> obj);;
val xobj : obj = 1

> let sobj = ("abc" :> obj);;
val sobj : obj = "abc"
```

Понижающий каст

F# Interactive

```
> let boxedObject = box "abc";;
val boxedObject : obj

> let downcastString = (boxedObject :?> string);;
val downcastString : string = "abc"

> let xobj = box 1;;
val xobj : obj = 1

> let x = (xobj :?> string);;
error: InvalidCastException raised at or near stdin:(2,0)
```

Каст и сопоставление шаблонов

```
let checkObject (x: obj) =
    match x with
    | :? string -> printfn "The object is a string"
    | :? int -> printfn "The object is an integer"
    | _ -> printfn "The input is something else"

let reportObject (x: obj) =
    match x with
    | :? string as s ->
        printfn "The input is the string '%s'" s
    | :? int as d ->
        printfn "The input is the integer '%d'" d
    | _ -> printfn "the input is something else"
```

Гибкие ограничения

F# Interactive

```
> open System.Windows.Forms
> let setTextOfControl (c : #Control) (s:string) =
    c.Text <- s;;
val setTextOfControl: #Control -> string -> unit

> open System.Windows.Forms;;
> let setTextOfControl (c : 'a when 'a :> Control)
    (s:string) = c.Text <- s;;
val setTextOfControl: #Control -> string -> unit
```

Гибкие ограничения: пример

```
module Seq =
...
val append : #seq<'a> -> #seq<'a> -> seq<'a>
val concat : #seq<#seq<'a>> -> seq<'a>
...

Seq.append [1; 2; 3] [4; 5; 6]
Seq.append [| 1; 2; 3 |] [4; 5; 6]
Seq.append (seq { for x in 1 .. 3 -> x }) [4; 5; 6]
Seq.append [| 1; 2; 3 |] [| 4; 5; 6 |]
```

Повышающий каст: проблема

```
open System
open System.IO
let textReader =
    if DateTime.Today.DayOfWeek = DayOfWeek.Monday
    then Console.In
    else File.OpenText("input.txt")
```

F# Interactive

```
else File.OpenText("input.txt")
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

error: FS0001: This expression has type StreamReader
but is here used with type TextReader
stopped due to error
```

Повышающий каст: решение

```
let textReader =
    if DateTime.Today.DayOfWeek = DayOfWeek.Monday
    then Console.In
    else (File.OpenText("input.txt") :> TextReader)
```

Проблемы в выводе типов, методы и свойства

F# Interactive

```
> let transformData inp =
    inp |> Seq.map (fun (x, y) -> (x, y.Length));;

inp |> Seq.map (fun (x, y) -> (x, y.Length))
---------------------------------------^^^^^^^^

stdin(11,36): error: Lookup on object of indeterminate
type. A type annotation may be needed prior to this
program point to constrain the type of the object.
This may allow the lookup to be resolved.
```

Решение

```
let transformData inp =
    inp |> Seq.map (fun (x, y:string) -> (x, y.Length))
```

Уменьшение общности

```
let printSecondElements (inp : #seq<'a * int>) =
    inp
    |> Seq.iter (fun (x, y) -> printfn "y = %d" x)
```

F# Interactive

```
|> Seq.iter (fun (x, y) -> printfn "y = %d" x)
-----------------------------------^^^^^^^^
stdin(21,38): warning: FS0064: This construct causes
code to be less generic than indicated by the type
annotations. The type variable 'a has been
constrained to the type 'int'.
```

Уменьшение общности, отладка

```
type PingPong = Ping | Pong

let printSecondElements (inp : #seq<PingPong * int>) =
    inp |> Seq.iter (fun (x, y) -> printfn "y = %d" x)
```

F# Interactive

```
|> Seq.iter (fun (x,y) -> printfn "y = %d" x)
-------------------------------------------^
stdin(27,47): error: FS0001: The type 'PingPong' is not
compatible with any of the types byte, int16, int32,
int64, sbyte, uint16, uint32, uint64, nativeint,
unativeint, arising from the use of a printf-style
format string
```

Value Restriction

F# Interactive

```
> let empties = Array.create 100 [];;
------^^^^^^^^
error: FS0030: Value restriction. Type inference
has inferred the signature
val empties : '_a list []
but its definition is not a simple data constant.
Either define 'empties' as a simple data expression,
make it a function, or add a type constraint
to instantiate the type parameters.
```

Корректные определения

```
let emptyList = []
let initialLists = ([], [2])
let listOfEmptyLists = [[]; []]
let makeArray () = Array.create 100 []
```

F# Interactive

```
val emptyList : 'a list
val initialLists : ('a list * int list)
val listOfEmptyLists : 'a list list
val makeArray : unit -> 'a list []
```

Способы борьбы

```
let empties = Array.create 100 []
let empties : int list [] = Array.create 100 []


let mapFirst = List.map fst
('a * 'b) list -> 'a list

let mapFirst inp = List.map fst inp
let printFstElements = List.map fst
    >> List.iter (printf "res = %d")

let printFstElements inp =
    inp
    |> List.map fst
    |> List.iter (printf "res = %d")
```

Point-free

```
let fstGt0 xs = List.filter (fun (a, b) -> a > 0) xs

let fstGt0'1 : (int * int) list -> (int * int) list =
    List.filter (fun (a, b) -> a > 0)

let fstGt0'2 : (int * int) list -> (int * int) list =
    List.filter (fun x -> fst x > 0)

let fstGt0'3 : (int * int) list -> (int * int) list =
    List.filter (fun x -> ((<) 0 << fst) x)

let fstGt0'4 : (int * int) list -> (int * int) list =
    List.filter ((<=) 0 << fst)
```

Арифметические операторы

F#

```
let twice x = (x + x)
let threeTimes x = (x + x + x)
let sixTimesInt64 (x:int64) = threeTimes x + threeTimes x
```

F# Interactive

```
val twice : x:int -> int
val threeTimes : x:int64 -> int64
val sixTimesInt64 : x:int64 -> int64
```

“За” и “против” ООП в функциональных языках

За:

- Портирование существующего кода
- Интеграция с другими языками
- Использование в основном для ООП с возможностью писать красивый код

Против:

- Не очень дружит с системой вывода типов
- Нет встроенной поддержки печати, сравнения и т.д.

Методы у типов

```
type Vector = {x : float; y : float} with
    member v.Length = sqrt(v.x * v.x + v.y * v.y)

let vector = {x = 1.0; y = 1.0}
let length = vector.Length

type Vector with
    member v.Scale k = {x = v.x * k; y = v.y * k}

let scaled = vector.Scale 2.0
```

Статические методы

F#

```
type Vector = {x : float; y : float} with
    static member Create x y = {x = x; y = y}

let vector = Vector.Create 1.0 1.0

type System.Int32 with
    static member IsEven x = x % 2 = 0

printfn "%b" <| System.Int32.IsEven 10
```

Каррирование против кортежей

```
type Vector with
    member v.TupledTransform (r, s) = transform v r s
    member v.CurriedTransform r s = transform v r s

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (45.0, 2.0)
printfn "%A" <| v.CurriedTransform 45.0 2.0
```

Кортежи: именованные аргументы

```
type Vector with
    member v.TupledTransform (r, s) =
        transform v r s

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (r = 45.0, s = 2.0)
printfn "%A" <| v.TupledTransform (s = 2.0, r = 45.0)
```

Кортежи: опциональные параметры

```
type Vector with
    member v.TupledTransform (r, ?s) =
        match s with
        | Some scale -> transform v r scale
        | None -> transform v r 1.0

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (45.0, 2.0)
printfn "%A" <| v.TupledTransform (90.0)
```

Кортежи: перегрузка

```
type Vector with
  member v.TupledTransform (r, s) =
    transform v r s
  member v.TupledTransform r =
    transform v r 1.0

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (45.0, 2.0)
printfn "%A" <| v.TupledTransform (90.0)
```

Кортежи против каррирования

За:

- Можно вызывать из .NET-кода
- Опциональные и именованные аргументы, перегрузки

Против:

- Не поддерживают частичное применение
- Не дружат с функциями высших порядков

Методы против свободных функций

```
type Vector = {x : float; y : float} with
    member v.Length = v.x * v.x + v.y * v.y |> sqrt

let length v = v.x * v.x + v.y * v.y |> sqrt

let compareWrong v1 v2 =
    v1.Length < v2.Length

let compareRight v1 v2 =
    length v1 < length v2
```

Классы, основной конструктор

```
type Vector(x, y) =
    member v.Length = x * x + y * y |> sqrt

printfn "%A" <| Vector (1.0, 1.0)
```

F# Interactive

```
FSI_0003+Vector
type Vector =
  class
    new : x:float * y:float -> Vector
    member Length : float
  end
val it : unit = ()
```

Методы и свойства

```
type Vector(x : float, y : float) =
  member v.Scale s = Vector(x * s, y * s)
  member v.X = x
  member v.Y = y
```

F# Interactive

```
type Vector =
 class
  new : x:float * y:float -> Vector
  member Scale : s:float -> Vector
  member X : float
  member Y : float
 end
```

Private-поля и private-методы

```
type Vector(x : float, y : float) =
    let mutable mX = x
    let mutable mY = y
    let lengthSqr = mX * mX + mY * mY
    member v.Length = sqrt lengthSqr
    member v.X = mX
    member v.Y = mY
    member v.SetX x = mX <- x
    member v.SetY y = mY <- y
```

Мутабельные свойства

```
type Vector(x, y) =
    let mutable mX = x
    let mutable mY = y
    member v.X
        with get () = mX
        and set x = mX <- x
    member v.Y
        with get () = mY
        and set y = mY <- y
```

Автоматические свойства

```
type Vector(x, y) =
    member val X = x with get,set
    member val Y = y with get,set

let v = Vector(1.0, 1.0)
v.X <- 2.0
```

Вернёмся к конструкторам

Дополнительное поведение

```
type Vector(x : float, y : float) =
    let length () = x * x + y * y |> sqrt
    do
        printfn "Vector (%f, %f), length = %f"
            x y <| length ()
        printfn "Have a nice day"
    let mutable x = x
    let mutable y = y

let v = Vector(1.0, 1.0)
```

Много конструкторов

```
type Vector(x : float, y : float) =
    member this.X = x
    member this.Y = y
    new () =
        printfn "Constructor with no parameters"
        Vector(0.0, 0.0)

let v = Vector(2.0, 2.0)
let v' = Vector()
```

Модификаторы видимости

```
type Example() =
    let mutable privateValue = 42

    member this.PublicValue = 1
    member private this.PrivateValue = 2
    member internal this.InternalValue = 3

    member this.PrivateSetProperty
        with get () =
            privateValue
        and private set(value) =
            privateValue <- value
```

Наследование

```
type Shape() =
    class
    end

type Circle(r) =
    inherit Shape()
    member this.R = r
```

Абстрактные классы

```
[<AbstractClass>]
type Shape() =
    abstract member Draw : unit -> unit
    abstract member Name : string

type Circle(r) =
    inherit Shape()
    member this.R = r
    override this.Draw () =
        printfn "Drawing circle"
    override this.Name = "Circle"
```

Реализация по умолчанию

```
type Shape() =
    abstract member Draw : unit -> unit
    abstract member Name : string
    default this.Draw () =
        printfn "Drawing shape"
    default this.Name =
        "Shape"
```

Интерфейсы

```
type Shape =
    abstract member Draw : unit -> unit
    abstract member Name : string

type Circle(r) =
    member this.R = r
    interface Shape with
        member this.Draw () =
            printfn "Drawing circle"
        member this.Name = "Circle"
```

Явное приведение типов

```
let c = Circle 10
c.Draw ()  // Ошибка
(c :> Shape).Draw ()  // Ок

let draw (s : Shape) = s.Draw ()

draw c  // Ок
```

Объектные выражения

Реализация интерфейсов на лету

```
type Shape =
    abstract member Draw : unit -> unit
    abstract member Name : string

let rect w h =
    { new Shape with
        member this.Draw () =
            printfn "Drawing rect, w = %d, h = %d" w h
        member this.Name = "Rectange"
    }

(rect 10 10).Draw ()
```

Модули

```
type Vector =
    { x : float; y : float }

module VectorOps =
    let length v = sqrt(v.x * v.x + v.y * v.y)
    let scale k v = { x = k * v.x; y = k * v.y }
    let shiftX x v = { v with x = v.x + x }
    let shiftY y v = { v with y = v.y + y }
    let shiftXY (x, y) v = { x = v.x + x; y = v.y + y }
    let zero = { x = 0.0; y = 0.0 }
    let constX dx = { x = dx; y = 0.0 }
    let constY dy = { x = 0.0; y = dy }
```

Пространства имён

```
namespace Vectors

type Vector =
    { x : float; y : float }

module VectorOps =
    let length v = sqrt(v.x * v.x + v.y * v.y)
```