Типы и генерики в F#

Юрий Литвинов
y.litvinov@spbu.ru

26.02.2026

Шаблонные типы

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

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

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)
```

Автоматический вывод типов

- Алгоритм Хиндли-Милнера
- На самом деле, «алгоритм *W* Дамаса-Милнера» над системой типов Хиндли-Милнера
  - Одно из типизированных λ-исчислений
  - Используется далеко не только в F#
- Построение системы уравнений над типами с учётом ограничений
  - литералы, функции и другие виды «взаимодействия значений», явные ограничения на типы, аннотации типов
- Решение методом унификации
  - Множество выражений и «переменных типа», им соответствующих
  - Постепенное уточнение этого множества
  - Если остались переменные типа, обобщение
  - Алгоритм глобальный!

Например

```
let outerFn action : string =
    let innerFn x = x + 1
    action (innerFn 2)
```

Тип посчитается в

```
val outerFn: (int -> string) -> string
```

А какой тип у функции ниже?

```
let doItTwice f  = (f >> f)
```

© https://fsharpforfunandprofit.com/posts/type-inference/

Однако не всё так просто

```
List.map (fun x -> x.Length) ["hello"; "world"]
```

— не скомпилируется, в момент вызова x неизвестно, что x строка

```
["hello"; "world"] |> List.map (fun x -> x.Length)
```

— скомпилируется

```
List.map (fun (x: string) -> x.Length) ["hello"; "world"]
```

— или так

Что ещё может пойти не так

```
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
```

— арифметические операторы не генерики!

Поэтому

```
let myNumericFn x = x * x
myNumericFn 10
myNumericFn 10.0  // Не скомпилится

let myNumericFn2 x = x * x
myNumericFn2 10.0
myNumericFn2 10  // Не скомпилится
```

Интересно, что такая типизация использовалась в языке Haxe (https://haxe.org/) для всего

Generic-сравнение

```
val (=) : 'a -> 'a -> bool
val (<) : 'a -> 'a -> bool
val (<=) : 'a -> 'a -> bool
val (>) : 'a -> 'a -> bool
val (>=) : 'a -> 'a -> bool
val compare : 'a -> 'a -> int
val min : 'a -> 'a -> 'a
val max : 'a -> 'a -> 'a
```

Сравнение сложных типов

F# Interactive

```
> ("abc", "def") < ("abc", "xyz");;
val it : bool = true
> compare (10, 30) (10, 20);;
val it : int = 1

> compare [10; 30] [10; 20];;
val it : int = 1
> compare [| 10; 30 |] [| 10; 20 |];;
val it : int = 1
> compare [| 10; 20 |] [| 10; 30 |];;
val it : int = -1
```

Generic-печать

F# Interactive

```
> sprintf "result = %A" ([1], [true]);;
val it : string = "result = ([1], [true])"

> sprintf "result = %O" ([1], [true]);;
val it : string = "result = ([1], [true])"
```

ToString():

```
> sprintf "result = %O" ([1], [true]);;
val it : string = "result = ([1], [true])"
```

Boxing/unboxing

F# Interactive

```
> box 1;;
val it : obj = 1

> box "abc";;
val it : obj = "abc"

> let sobj = box "abc";;
val sobj : obj = "abc"

> (unbox<string> sobj);;
val it : string = "abc"

> (unbox sobj : string);;
val it : string = "abc"
```

Сериализация

```
open System.IO
open System.Runtime.Serialization.Formatters.Binary

let writeValue outputStream (x: 'a) =
    let formatter = new BinaryFormatter()
    formatter.Serialize(outputStream, box x)

let readValue inputStream =
    let formatter = new BinaryFormatter()
    let res = formatter.Deserialize(inputStream)
    unbox res
```

Сериализация, пример использования

```
let addresses = Map.ofList [
    "Jeff", "123 Main Street, Redmond, WA 98052";
    "Fred", "987 Pine Road, Phila., PA 19116";
    "Mary", "PO Box 112233, Palo Alto, CA 94301" ]

use fsOut = new FileStream("Data.dat", FileMode.Create)
writeValue fsOut addresses
fsOut.Close()

use fsIn = new FileStream("Data.dat", FileMode.Open)
let res : Map<string, string> = readValue fsIn
fsIn.Close()
```

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
```

Тип функции

F# Interactive

```
val hcfGeneric : Numeric<'a> -> ('a -> 'a -> 'a)
```

Примеры использования

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

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

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

Результат

F# Interactive

```
val hcfInt : (int -> int -> int)
val hcfBigInt : (bigint -> bigint -> bigint)

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

> hcfBigInt 181028711616223238303957 6I
    123902817829309283048023903 2I;;
val it : bigint = 33224I
```

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 : 'a when 'a :> Control)
    (s:string) = c.Text <- s;;
val setTextOfControl: #Control -> string -> unit

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

Гибкие ограничения: пример

```
let iterate1 (f : unit -> seq<int>) =
    for e in f() do printfn "%d" e
let iterate2 (f : unit -> #seq<int>) =
    for e in f() do printfn "%d" e

// Passing a function that takes a list requires a cast.
iterate1 (fun () -> [1] :> seq<int>)

// Passing a function that takes a list to the version that specifies a
// flexible type as the return value is OK as is.
iterate2 (fun () -> [1])
```

© https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/flexible-types

Примеры

```
let list1 = [ 1; 2; 3 ]
let list2 = [ 4; 5; 6 ]
let list3 = [ 7; 8; 9 ]

let concat1 = Seq.concat [ list1; list2; list3]
printfn "%A" concat1
```

© https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/flexible-types

Примеры (2)

```
let array1 = [| 1; 2; 3 |]
let array2 = [| 4; 5; 6 |]
let array3 = [| 7; 8; 9 |]

let concat2 = Seq.concat [ array1; array2; array3 ]
printfn "%A" concat2

let concat3 = Seq.concat [| list1; list2; list3 |]
printfn "%A" concat3
```

© https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/flexible-types

Примеры (3)

```
let concat4 = Seq.concat [| array1; array2; array3 |]
printfn "%A" concat4

let seq1 = { 1 .. 3 }
let seq2 = { 4 .. 6 }
let seq3 = { 7 .. 9 }

let concat5 = Seq.concat [| seq1; seq2; seq3 |]

printfn "%A" concat5
```

© https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/flexible-types

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 (1)

Явная аннотация типа (не генерик):

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

Способы борьбы (2)

```
let mapFirst = List.map fst
```

— не скомпилится, тип посчитается в

`('_a * '_b) list -> '_a list`

Сделать из значения функцию:

```
let mapFirst inp = List.map fst inp
```

Способы борьбы (3)

Более хитрый пример:

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

выведется в

```
((int * '_a) list -> unit)
```

— недообобщённый тип _a. Чинится η-преобразованием:

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

Способы борьбы (4)

Вынесение явного параметра-типа:

```
let emptyLists = Seq.init 100 (fun _ -> [])
```

— не скомпилится

```
let emptyLists<'a> : seq<'a list> = Seq.init 100 (fun _ -> [])
```

— скомпилится, это на самом деле функция над параметром-типом и компилируется в генерик-метод!

А именно

F# Interactive

```
> Seq.length emptyLists;;
val it : int = 100

> emptyLists<int>;;
val it : seq<int list> = seq [[]; []; []; []; ...]

> emptyLists<string>;;
val it : seq<string list> = seq [[]; []; []; []; ...]
```

Подробности: https://habr.com/ru/company/microsoft/blog/348460/

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)
```