

Вычислительные выражения в F#

Computation Expressions, Workflows

Юрий Литвинов

23.12.2016г

Что это и зачем нужно

«a monad is a monoid in the category of endofunctors, what's the problem?»

- ▶ Механизм управления процессом вычислений
- ▶ В функциональных языках — единственный способ определить порядок вычислений
- ▶ Зачастую — нетривиальным образом (Async)
- ▶ Способ не писать кучу вспомогательного кода (сродни АОП)
- ▶ В теории ФП они называются монадами
- ▶ На самом деле, синтаксический сахар

Пример

Классический пример с делением на 0

Сопротивление сети из параллельных резисторов:

$$1/R = 1/R_1 + 1/R_2 + 1/R_3$$

R_1 , R_2 и R_3 могут быть 0. Что делать?

- ▶ Бросать исключение — плохо
- ▶ Использовать option — много работы, но попробуем

Реализация вручную

divide

F#

```
let divide x y =  
    match y with  
    | 0.0 -> None  
    | _ -> Some (x / y)
```

Реализация вручную

Само вычисление

F#

```
let resistance r1 r2 r3 =  
    let r1' = divide 1.0 r1  
    match r1' with  
    | None -> None  
    | Some x -> let r2' = divide 1.0 r2  
                match r2' with  
                | None -> None  
                | Some y -> let r3' = divide 1.0 r3  
                            match r3' with  
                            | None -> None  
                            | Some z -> let r = divide 1.0 (x + y + z)  
                                        r
```

То же самое, через Workflow Builder

F#

```
type MaybeBuilder() =  
    member this.Bind(x, f) =  
        match x with  
        | None -> None  
        | Some a -> f a  
    member this.Return(x) =  
        Some x  
  
let maybe = new MaybeBuilder()
```

Само вычисление

F#

```
let resistance r1 r2 r3 =  
    maybe {  
        let! r1' = divide 1.0 r1  
        let! r2' = divide 1.0 r2  
        let! r3' = divide 1.0 r3  
        let! r = divide 1.0 (r1' + r2' + r3')  
        return r  
    }
```

Некоторые синтаксические "похожести"

seq — это тоже Computation Expression

F#

```
let daysOfTheYear =  
    seq {  
        let months =  
            [ "Jan"; "Feb"; "Mar"; "Apr"; "May"; "Jun";  
              "Jul"; "Aug"; "Sep"; "Oct"; "Nov"; "Dec" ]  
        let daysInMonth month =  
            match month with  
            | "Feb" -> 28  
            | "Apr" | "Jun" | "Sep" | "Nov" -> 30  
            | _ -> 31  
        for month in months do  
            for day = 1 to daysInMonth month do  
                yield (month, day)  
    }
```


Ещё один пример

F#

```
let debug x = printfn "value is %A" x
```

```
let withDebug =  
    let a = 1  
    debug a  
    let b = 2  
    debug b  
    let c = a + b  
    debug c  
    c
```

То же самое с Workflow

F#

```
type DebugBuilder() =  
    member this.Bind(x, f) =  
        debug x  
        f x  
    member this.Return(x) = x  
  
let debugFlow = DebugBuilder ()  
  
let withDebug = debugFlow {  
    let! a = 1  
    let! b = 2  
    let! c = a + b  
    return c  
}
```

Что происходит

Как оно устроено внутри

- ▶ Bind создаёт цепочку continuation passing style-функций, возможно, с побочными эффектами
- ▶ Есть тип-обёртка (или монадический тип), в котором хранится состояние вычисления
- ▶ let! вызывает Bind, return — Return, Bind принимает обёрнутое значение и функцию-continuation, return по необёрнутому значению делает обёрнутое

Отступление про CPS

C#, код без CPS

C#

```
public int Divide(int a, int b)
{
    if (b == 0)
    {
        throw new InvalidOperationException("div by 0");
    }
    else
    {
        return a / b;
    }
}
```

C#, то же с CPS

C#

```
public T Divide<T>(int a, int b, Func<T> ifZero
    , Func<int, T> ifSuccess)
{
    if (b == 0)
    {
        return ifZero();
    }
    else
    {
        return ifSuccess(a / b);
    }
}
```

Вызывающий решает, что делать, а не вызываемый.

То же на F#

Без CPS:

F#

```
let divide a b =  
    if (b = 0)  
    then invalidOp "div by 0"  
    else (a / b)
```

C CPS:

F#

```
let divide ifZero ifSuccess a b =  
    if (b = 0)  
    then ifZero()  
    else ifSuccess (a / b)
```

Примеры

F#

```
let ifZero1 () = printfn "bad"
let ifSuccess1 x = printfn "good %i" x
let divide1 = divide ifZero1 ifSuccess1

let ifZero2 () = None
let ifSuccess2 x = Some x
let divide2 = divide ifZero2 ifSuccess2

let ifZero3 () = failwith "div by 0"
let ifSuccess3 x = x
let divide3 = divide ifZero3 ifSuccess3
```

let, «многословный» синтаксис

F#

```
let x = something
```

равносильно

F#

```
let x = something in [ выражение с x ]
```

например,

F#

```
let x = 1 in
  let y = 2 in
    let z = x + y in
      z
```


let и лямбды

F#

```
fun x -> [ выражение с x ]
```

или

F#

```
something |> (fun x -> [ выражение с x ])
```

и обращаем внимание, что:

F#

```
let x = someExpression in [ выражение с x ]  
someExpression |> (fun x -> [ выражение с x ])
```

let и CPS

F#

```
let x = 1 in
    let y = 2 in
        let z = x + y in
            z
```

F#

```
1 |> (fun x ->
2 |> (fun y ->
    x + y |> (fun z ->
        z)))
```

Теперь вспомним про Workflow-ы

F#

```
let pipeInto expr f  
    expr |> f
```

F#

```
pipeInto (1, fun x ->  
    pipeInto (2, fun y ->  
        pipeInto (x + y, fun z ->  
            z)))
```

Зачем

F#

```
let pipeInto (expr, f) =  
    printfn "expression is %A" expr  
    expr |> f
```

F#

```
pipeInto (1, fun x ->  
    pipeInto (2, fun y ->  
        pipeInto (x + y, fun z ->  
            z)))
```

То же самое с Workflow

F#

```
type DebugBuilder() =  
    member this.Bind(x, f) =  
        debug x  
        f x  
    member this.Return(x) = x  
  
let debugFlow = DebugBuilder ()  
  
let withDebug = debugFlow {  
    let! a = 1  
    let! b = 2  
    let! c = a + b  
    return c  
}
```

Более сложный пример, с делением

pipeInto, которая потом будет Bind

F#

```
let pipeInto (expr, f) =  
    match expr with  
    | None ->  
        None  
    | Some x ->  
        x |> f
```

Более сложный пример, с делением

Сам процесс

F#

```
let resistance r1 r2 r3 =  
    let a = divide 1.0 r1  
    pipeInto (a, fun a' ->  
        let b = divide 1.0 r2  
        pipeInto (b, fun b' ->  
            let c = divide 1.0 r3  
            pipeInto (c, fun c' ->  
                let r = divide 1.0 (a + b + c)  
                pipeInto (r, fun r' ->  
                    Some r  
                ))))
```

Уберём временные let-ы

F#

```
let resistance r1 r2 r3 =  
    pipeInto (divide 1.0 r1, fun a ->  
        pipeInto (divide 1.0 r2, fun b ->  
            pipeInto (divide 1.0 r3, fun c ->  
                pipeInto (divide 1.0 (a + b + c), fun r ->  
                    Some r  
                )))  
        )))
```


И отформатируем

F#

```
let resistance r1 r2 r3 =  
    pipeInto (divide 1.0 r1, fun a ->  
    pipeInto (divide 1.0 r2, fun b ->  
    pipeInto (divide 1.0 r3, fun c ->  
    pipeInto (divide 1.0 (a + b + c) , fun r ->  
    Some r  
    ))))
```

Сравним с оригиналом

F#

```
let resistance r1 r2 r3 =  
    maybe {  
        let! r1' = divide 1.0 r1  
        let! r2' = divide 1.0 r2  
        let! r3' = divide 1.0 r3  
        let! r = divide 1.0 (r1' + r2' + r3')  
        return r  
    }
```

Подробнее про Bind

- ▶ $\text{Bind} : M<'T> * ('T \rightarrow M<'U>) \rightarrow M<'U>$
- ▶ $\text{Return} : 'T \rightarrow M<'T>$

F#

```
let! x = 1 in x * 2
```

F#

```
builder.Bind(1, (fun x -> x * 2))
```

Инфиксное определение Bind

F#

```
let (>>=) m f = pipeInto(m, f)
```

```
let workflow =  
    1 >>= (+) 2 >>= (*) 42 >>= id
```

Option.bind и maybe

F#

```
module Option =  
    let bind f m =  
        match m with  
        | None ->  
            None  
        | Some x ->  
            x |> f  
  
type MaybeBuilder() =  
    member this.Bind(m, f) = Option.bind f m  
    member this.Return(x) = Some x
```

Содержимое типа-обёртки может иметь разный тип

Пример, серия запросов к БД

F#

```
type DbResult<'a> =  
    | Success of 'a  
    | Error of string
```

```
type CustomerId = CustomerId of string  
type OrderId = OrderId of int  
type ProductId = ProductId of string
```

Пример, запросы

F#

```
let getCustomerId name =  
    if (name = "")  
    then Error "getCustomerId failed"  
    else Success (CustomerId "Cust42")  
  
let getLastOrderForCustomer (CustomerId custId) =  
    if (custId = "")  
    then Error "getLastOrderForCustomer failed"  
    else Success (OrderId 123)  
  
let getLastProductForOrder (OrderId orderId) =  
    if (orderId = 0)  
    then Error "getLastProductForOrder failed"  
    else Success (ProductId "Product456")
```

Общение с БД вручную

F#

```
let product =  
    let r1 = getCustomerId "Alice"  
    match r1 with  
    | Error e -> Error e  
    | Success custId ->  
        let r2 = getLastOrderForCustomer custId  
        match r2 with  
        | Error e -> Error e  
        | Success orderId ->  
            let r3 = getLastProductForOrder orderId  
            match r3 with  
            | Error e -> Error e  
            | Success productId ->  
                printfn "Product is %A" productId  
                r3
```


Builder

F#

```
type DbResultBuilder() =  
  
    member this.Bind(m, f) =  
        match m with  
        | Error e -> Error e  
        | Success a ->  
            printfn "Successful: %A" a  
            f a  
  
    member this.Return(x) =  
        Success x  
  
let dbresult = new DbResultBuilder()
```

Workflow

F#

```
let product =  
    dbresult {  
        let! custId = getCustomerId "Alice"  
        let! orderId = getLastOrderForCustomer custId  
        let! productId = getLastProductForOrder orderId  
        printfn "Product is %A" productId  
        return productId  
    }  
    printfn "%A" product
```

Композиция Workflow-ов

F#

```
let subworkflow1 = myworkflow { return 42 }
let subworkflow2 = myworkflow { return 43 }

let aWrappedValue =
    myworkflow {
        let! unwrappedValue1 = subworkflow1
        let! unwrappedValue2 = subworkflow2
        return unwrappedValue1 + unwrappedValue2
    }
```

Вложенные Workflow-ы

F#

```
let aWrappedValue =  
    myworkflow {  
        let! unwrappedValue1 = myworkflow {  
            let! x = myworkflow { return 1 }  
            return x  
        }  
        let! unwrappedValue2 = myworkflow {  
            let! y = myworkflow { return 2 }  
            return y  
        }  
        return unwrappedValue1 + unwrappedValue2  
    }
```

ReturnFrom

F#

```
type MaybeBuilder() =  
    member this.Bind(m, f) = Option.bind f m  
    member this.Return(x) =  
        printfn "Wrapping a raw value into an option"  
        Some x  
    member this.ReturnFrom(m) =  
        printfn "Returning an option directly"  
        m  
  
let maybe = new MaybeBuilder()
```

Пример

F#

```
maybe { return 1 }
```

```
maybe { return! (Some 2) }
```

Зачем это

F#

```
maybe {  
    let! x = divide 24 3  
    let! y = divide x 2  
    return y  
}
```

```
maybe {  
    let! x = divide 24 3  
    return! divide x 2  
}
```

Первый закон монад

- Bind и Return должны быть взаимно обратны

F#

```
myworkflow {  
    let originalUnwrapped = something  
    let wrapped = myworkflow { return originalUnwrapped }  
    let! newUnwrapped = wrapped  
    assertEquals newUnwrapped originalUnwrapped  
}  
  
myworkflow {  
    let originalWrapped = something  
    let newWrapped = myworkflow {  
        let! unwrapped = originalWrapped  
        return unwrapped  
    }  
    assertEquals newWrapped originalWrapped  
}
```


Второй закон монад

- Композиция должна быть консистентной

F#

```
let result1 = myworkflow {  
    let! x = originalWrapped  
    let! y = f x  
    return! g y  
}  
let result2 = myworkflow {  
    let! y = myworkflow {  
        let! x = originalWrapped  
        return! f x  
    }  
    return! g y  
}  
assertEqual result1 result2
```

Какие ещё методы есть у WorkflowBuilder

Имя	Тип	Описание
Delay	$(\text{unit} \rightarrow M<'T>) \rightarrow M<'T>$	Превращает в функцию
Run	$M<'T> \rightarrow M<'T>$	Исполняет вычисление
Combine	$M<'T> * M<'T> \rightarrow M<'T>$	Последовательное исполнение
For	$\text{seq}<'T> * ('T \rightarrow M<'U>) \rightarrow M<'U>$	Цикл for
TryWith	$M<'T> * (\text{exn} \rightarrow M<'T>) \rightarrow M<'T>$	Блок try with
TryFinally	$M<'T> * (\text{unit} \rightarrow \text{unit}) \rightarrow M<'T>$	Блок finally
Using	$'T * ('T \rightarrow M<'U>) \rightarrow M<'U>$ when 'U := IDisposable	use
While	$(\text{unit} \rightarrow \text{bool}) * M<'T> \rightarrow M<'T>$	Цикл while
Yield	$'T \rightarrow M<'T>$	yield или ->
YieldFrom	$M<'T> \rightarrow M<'T>$	yield! или ->>
Zero	$\text{unit} \rightarrow M<'T>$	Обёрнутое ()

Моноиды

Немного алгебры

Множество с бинарной операцией

- ▶ Замкнутость относительно операции
- ▶ Ассоциативность
- ▶ Наличие нейтрального элемента

Например, $[a] @ [b] = [a; b]$

Пример

F#

```
type OrderLine = { Quantity : int; Total : float }
```

```
let orderLines = [  
    { Quantity = 2; Total = 19.98 };  
    { Quantity = 1; Total = 1.99 };  
    { Quantity = 2; Total = 3.98 }; ]
```

```
let addLine line1 line2 =  
    { Quantity = line1.Quantity + line2.Quantity;  
      Total = line1.Total + line2.Total }
```

```
orderLines |> List.reduce addLine
```

Эндоморфизмы

Эндоморфизм — функция, у которой тип входного значения совпадает с типом выходного

Множество функций + композиция — моноид, если функции — эндоморфизмы

Пример

F#

```
let plus1 x = x + 1
let times2 x = x * 2
let subtract42 x = x - 42

let functions = [
    plus1;
    times2;
    subtract42 ]

let newFunction = functions |> List.reduce (>>)

printfn "%d" <| newFunction 20
```

Bind

`Option.bind : ('T → 'U option) → 'T option → 'U option`

— частично применённый Bind — эндоморфизм (если 'T и 'U совпадают)

F#

```
let bindFns = [  
    Option.bind (fun x -> if x > 1 then  
                          Some (x * 2)  
                          else None);  
    Option.bind (fun x -> if x < 10 then Some x else None)  
]
```

```
let bindAll =  
    bindFns |> List.reduce (>>)
```

```
Some 4 |> bindAll
```

Не только эндоморфизмы могут образовать моноид

F#

```
type Predicate <'A> = 'A -> bool
```

```
let predAnd p1 p2 x =  
    if p1 x  
    then p2 x  
    else false
```

```
let predicates = [isMoreThan10Chars; isMixedCase; isNotDicti
```

```
let combinePredicates = predicates |> List.reduce predAnd
```


Монады

Workflow-ы, Computational Expressions

- ▶ Замкнуты
- ▶ Композиция ассоциативна (второй закон монад)
- ▶ Нейтральный элемент (Return, первый закон монад)

«a monad is a monoid in the category of endofunctors, what's the problem?»

Полезные ссылки

Откуда взяты примеры

- ▶ <https://fsharpforfunandprofit.com/series/computation-expressions.html>
- ▶ <http://www.slideshare.net/ScottWlaschin/fp-patterns-buildstuffit>

Пример: Async workflow

F#

```
open System.Net
open System.IO
let sites = ["http://se.math.spbu.ru";
             "http://spisok.math.spbu.ru"]
let fetchAsync url =
    async {
        do printfn "Creating request for %s..." url
        let request = WebRequest.Create(url)
        use! response = request.AsyncGetResponse()
        do printfn "Getting response stream for %s..." url
        use stream = response.GetResponseStream()
        do printfn "Reading response for %s..." url
        use reader = new StreamReader(stream)
        let html = reader.ReadToEnd()
        do printfn "Read %d characters for %s..." html.Length url
    }

sites |> List.map (fun site -> site |> fetchAsync |> Async.Start)
|> ignore
```

Что получится

F# Interactive

```
Creating request for http://se.math.spbu.ru ...
Creating request for http://spisok.math.spbu.ru ...
val sites : string list =
    ["http://se.math.spbu.ru"; "http://spisok.math.spbu.ru"]
val fetchAsync : url:string -> Async<unit>
val it : unit = ()

> Getting response stream for http://spisok.math.spbu.ru ...
Reading response for http://spisok.math.spbu.ru ...
Read 4475 characters for http://spisok.math.spbu.ru ...
Getting response stream for http://se.math.spbu.ru ...
Reading response for http://se.math.spbu.ru ...
Read 217 characters for http://se.math.spbu.ru ...
```

Использование пула потоков

Распечатаем Id потоков, в которых вызываются методы printfn:

```
F#  
  
open System.Threading  
  
let tprintfn fmt =  
    printf "[%NET Thread %d]"  
        Thread.CurrentThread.ManagedThreadId;  
    printfn fmt
```

Что получилось теперь

F# Interactive

```
[.NET Thread 47][.NET Thread 49]Creating request
    for http://se.math.spbu.ru ...
Creating request for http://spisok.math.spbu.ru ...
val sites : string list =
    ["http://se.math.spbu.ru"; "http://spisok.math.spbu.ru"]
val tprintfn : fmt:Printf.TextWriterFormat<'a> -> 'a
val fetchAsync : url:string -> Async<unit>
val it : unit = ()

> [.NET Thread 49]Getting response stream for
    http://spisok.math.spbu.ru ...
[.NET Thread 49]Reading response for http://spisok.math.spbu.ru ...
[.NET Thread 50]Getting response stream for http://se.math.spbu.ru ...
[.NET Thread 50]Reading response for http://se.math.spbu.ru ...
[.NET Thread 50][.NET Thread 49]Read 217 characters
    for http://se.math.spbu.ru ...
Read 4475 characters for http://spisok.math.spbu.ru ...
```

Какие конструкции поддерживает Async

Конструкция	Описание
<code>let! pat = expr</code>	Выполняет асинхронное вычисление <code>expr</code> и присваивает результат <code>pat</code> , когда оно заканчивается
<code>let pat = expr</code>	Выполняет синхронное вычисление <code>expr</code> и присваивает результат <code>pat</code> немедленно
<code>use! pat = expr</code>	Выполняет асинхронное вычисление <code>expr</code> и присваивает результат <code>pat</code> , когда оно заканчивается. Вызовет <code>Dispose</code> для каждого имени из <code>pat</code> , когда <code>Async</code> закончится.
<code>use pat = expr</code>	Выполняет синхронное вычисление <code>expr</code> и присваивает результат <code>pat</code> немедленно. Вызовет <code>Dispose</code> для каждого имени из <code>pat</code> , когда <code>Async</code> закончится.
<code>do! expr</code>	Выполняет асинхронную операцию <code>expr</code> , эквивалентно <code>let! () = expr</code>
<code>do expr</code>	Выполняет синхронную операцию <code>expr</code> , эквивалентно <code>let () = expr</code>
<code>return expr</code>	Оборачивает <code>expr</code> в <code>Async<'T></code> и возвращает его как результат <code>Workflow</code>
<code>return! expr</code>	Возвращает <code>expr</code> типа <code>Async<'T></code> как результат <code>Workflow</code>

Control.Async

Что можно делать со значением Async<'T>, сконструированным билдером

Метод	Тип	Описание
RunSynchronously	Async<'T> * ?int * ?CancellationTokens -> 'T	Выполняет вычисление синхронно, возвращает результат
Start	Async<unit> * ?CancellationTokens -> unit	Запускает вычисление асинхронно, тут же возвращает управление
Parallel	seq<Async<'T> > -> Async<'T []>	По последовательности Async-ов делает новый Async, исполняющий все Async-и параллельно и возвращающий массив результатов
Catch	Async<'T> -> Async<Choice<'T,exn> >	По Async-у делает новый Async, исполняющий Async и возвращающий либо результат, либо исключение

Пример

F#

```
let writeFile fileName bufferData =  
    async {  
        use outputFile = System.IO.File.Create(fileName)  
        do! outputFile.AsyncWrite(bufferData)  
    }  
  
Seq.init 1000 (fun num -> createSomeData num)  
|> Seq.mapi (fun num value ->  
    writeFile ("file" + num.ToString() + ".dat") value)  
|> Async.Parallel  
|> Async.RunSynchronously  
|> ignore
```

Отмена операции

Код, который её отменяет

F#

```
let cancelHandler (ex : OperationCanceledException) =  
    printfn "The task has been canceled."
```

```
Async.TryCancelled(cancelableTask, cancelHandler)  
|> Async.Start
```

```
// ...
```

```
Async.CancelDefaultToken()
```