ITT8060
Advanced Programming

In F#

Lecture 11: Computation Expressions

Juhan Ernits, Hendrik Maarand
and Ed Morehouse

New syntax introduced in the FsCheck examples

Generators

- „Generators have types of the form `Gen<'a>;` this is a generator for values of type `a`. To build your own generators in F#, a computation expression called `gen` is provided by FsCheck "

Domain Specific Language for Generators

```
val chooseFromList : xs:'a list -> Gen<'a>
```

```
let chc
    gen {

        let! idx = Gen.choose (0, List.length xs -
1)
         return (List.item idx xs)
    }
```

Domain Specific Language for Generators

```
let cho
```

```
val chooseFromList : xs:'a list -> Gen<'a>
```

```
    gen {

        let! idx = Gen.choose (0, List.length xs -
1)
        return (List.item idx xs)
    }
```

```
val chooseFromList' : xs:'a list -> Gen<'a>
```

```
let chooseFromList' xs =
    gen.Bind(
        (Gen.choose (0, List.length xs - 1),
            (fun idx ->
                gen.Return (List.item idx xs))))
```

Computation Expressions

Computation expressions and monads

- Computation expressions are the F# equivalent of monadic syntax in Haskell
- Monads are a powerful design pattern:
  - characterized by type M<’T> and combined with at least 2 operators:
    - `bind: M<’T> -> (’T -> M< ’U>) -> M<’U>`
    - `return: ’T -> M< ’T>`

Computation expressions and monads

- Computation expressions are the F# equivalent of monadic syntax in Haskell
- Monads are a powerful design pattern:
  - characterized by type M<'T> and combined with at least 2 operators:

    `let!`

    - `bind: M<'T> -> ('T -> M< 'U>) -> M<'U>`
    - `return: 'T -> M< 'T>`

    `return`

F# computation expressions

- Are inspired by Haskell **monads**
- Are slightly different from Haskell monads as F# combined with imperative programming can have **side effects not tracked by the type system**
- F# computation expressions can also be used to embed computations that generate multiple results (also known as **monoids**).
  - E.g. **sequence expressions** (also known as comprehension syntax)
  - Use `yield` and `yield!` instead of `return` and `return!` and often do not have `let!`

Computation expressions

- Syntactic extension to F# to help define “non-standard” computations in a compositional, uniform way
- Syntax:

```
builder { computation-expression }
```

Sequence expressions

```
seq {
      for i in 1..9 do
        for j in 1..9 do
          yield (i, j, i*j)
      }
```

Unfold dynamically

Asynchronous workflows

Asynchronous "non-blocking" action

```
async {
    let! image = readAsync "cat.jpg"
    let image2 = f image
    do! writeAsync image2 "dog.jpg"
    do printfn "done!"
    return image2
}
```

"Thread" continuation/
Event callback

Syntax

expr :=	
expr { cexpr }	Computation expression
cexpr :=	
let! pat = expr in cexpr	Binding computation
do! expr in cexpr	Sequential computation
use! pat = expr in cexpr	Auto clean-up computation
yield! expr	Yield computation
yield expr	Yield result
return! expr	Return computation
return expr	Return result
for pat in expr do expr	Yield result

De-sugaring

expr { cexpr }	$\Rightarrow$ let b = expr b.Run (b.Delay (fun () -> {| cexpr |}$_c$))
{| let! pat = expr in cexpr |}$_c$	$\Rightarrow$ b.Bind (expr, (fun pat -> {| cexpr |}$_c$))
{| do! expr in cexpr |}$_c$	$\Rightarrow$ b.Bind (expr, (fun () -> {| cexpr |}$_c$))
{| do expr in cexpr |}$_c$	$\Rightarrow$ expr; {| cexpr |}$_c$
{| yield! expr |}$_c$	$\Rightarrow$ b.YieldFrom (expr)
{| yield expr |}$_c$	$\Rightarrow$ b.Yield (expr)
{| return! expr |}$_c$	$\Rightarrow$ b.ReturnFrom (expr)
{| return expr |}$_c$	$\Rightarrow$ b.Return(expr)
{| for pat in expr do expr |}$_c$	$\Rightarrow$ b.For({|expr|}$_E$, (fun pat -> {|expr |}$_c$))

De-sugaring

expr { cexpr } $\Rightarrow$ let b = expr
b.Run (b.Delay (fun () -> {| cexpr |}$_c$))

{| let! pat = expr in cexpr |}$_c$ $\Rightarrow$ b.Bind (expr, (fun pat -> {| cexpr |}$_c$))
{| return expr |}$_c$ $\Rightarrow$ b.Return(expr)

example

```
async {
    let! x = AsyncRead ()
    return x
}
```

$\Rightarrow$

```
let b = async
b.Delay (fun () ->
b.Bind(AsyncRead(), fun x ->
b.Return x))
```

Async<'a>

'a -> Async<'a>

'a -> Async<'a>

Type AsyncBuilder

```
type AsyncBuilder =
class
  new AsyncBuilder : unit -> AsyncBuilder
  member this.Bind: Async<‘a> * (‘a -> Async <‘b>) -> Async <‘b>
  member this.Return: ‘a -> Async <‘a>
  member this.Delay: (unit -> Async <‘a>) -> Async <‘a>
end
```

example

```
async {
    let! x = AsyncRead ()
    return x
}
⇒
let b = async
b.Delay (fun () ->
b.Bind(AsyncRead(), fun x ->
b.Return x))
```

Async<‘a>

‘a -> Async<‘a>

‘a -> Async<‘a>

Type AsyncBuilder

```
type AsyncBuilder =
class
  new AsyncBuilder : unit -> AsyncBuilder
  member this.Bind: Async<‘a> * (‘a -> Async <‘b>) -> Async <‘b>
  member this.Return: ‘a -> Async <‘a>
  member this.Delay: (unit -> Async <‘a>) -> Async <‘a>
end
```

By definition, M is a monad when the following operations are given:

- Bind: M<‘a> -> (‘a -> M<‘b>) -> M<‘b>
- Return : ‘a -> M<‘a>

Example 1: Logging workflow

```
let write s =
    logging {
        let! _ = log ("writing: " + s)
        System.Console.WriteLine s
    }

let read () =
    logging {
        do! log "reading"
        return Console.ReadLine()
    }

let loggingTest() =
    logging {
        do! log "starting"
        do! write "Enter name: "
        let! name = read ()
        return "Hello " + name + "!"
    }

loggingTest()

Enter name:
Nimi
val it : string Logging =
  Logging ("Hello Nimi!", ["starting"; "writing: Enter name: "; "reading"])
```

```
type 'a Logging =
    | Logging of 'a * string list

type LoggingBuilder () =
    member this.Bind (Logging(x,msgs1), f) =
        let (Logging (y, msgs2)) = f x
        Logging (y, msgs1 @ msgs2)
    member this.Return x =
        Logging (x,[])
    member this.Zero () =
        Logging ((),[])
```

From Programming F#

Example 2: The Rounding Workflow

```
let x = 2.0 / 12.0
let y = 3.5
x/y
```

```
val it : float = 0.04761904762
```

```
withPrecision 5 {
   let! x = 2.0 / 12.0
   let! y = 3.5
   return x / y
}
```

```
val it : float = 0.04762
```

```
type RoundingWorkflow(sigDigs:int) =
   let round (x:float) = System.Math.Round(float x, sigDigs)
   member this.Bind(result:float, rest:float->float) = rest (round result)
   member this.Return(x:float) = round x

let withPrecision sigDigs = new RoundingWorkflow(sigDigs)
```

https://mbrace.io

Example 3: MBrace training in cloud

Part 1 - Extract Statistics in the Cloud

```
#load "../lib/utils.fsx"

// Initialize client object to an MBrace cluster
let cluster = Config.GetCluster()
let fs = cluster.Store.CloudFileSystem
```

Step 1: download text file from source, saving it to blob storage chunked into smaller files of 10000 lines each.

```
let download (uri: string) =
    cloud {
        let webClient = new WebClient()
        do! Cloud.Log "Begin file download"
        let text = webClient.DownloadString(uri)
        do! Cloud.Log "file downloaded"
        // Partition the big text into smaller files
        let! files =
            text.Split('\n')
            |> Array.chunkBySize 10000
            |> Array.mapi (fun index lines ->
                 local {
                    fs.File.Delete(sprintf "text/%d.txt" index)
                    let file = fs.File.WriteAllLines(path = sprintf "text/%d.txt" index, lines = lines)
                    return file })
            |> Local.Parallel
        return files
    }

let downloadTask = download "http://norvig.com/big.txt" |> cluster.CreateProcess

downloadTask.ShowInfo()

let files = downloadTask.Result

(** Now, take a look at the sizes of the files. *)
let fileSizesJob =
    files
    |> Array.map (fun f -> CloudFile.GetSize f.Path)
    |> Cloud.ParallelBalanced
    |> cluster.CreateProcess

fileSizesJob.Status
fileSizesJob.ShowInfo()

let fileSizes = fileSizesJob.Result
```

Example 4: F# for the web (flows)

- WebSharper (http://websharper.com/)

```
type Person =
    {
        Name : string
        Address : string
    }

type ContactType = | EmailTy | PhoneTy

type ContactDetails =
    | Email of string
    | PhoneNumber of string
```

To create a flowlet, we use the `Define` function.

```
val Define : (('A -> unit) -> Doc) -> Flow<'A>
```

```
let personFlowlet =
    Flow.Define (fun cont ->
        let rvName = Var.Create ""
        let rvAddress = Var.Create ""

        Doc.Input [] rvName
        Doc.Input [] rvAddress
        Doc.Button "Next" [cls "btn" ; cls "btn-default"] (fun () ->
            let name = Var.Get rvName
            let addr = Var.Get rvAddress
            // We use the continuation function to return the
            // data we retrieved from the form.
            cont ({Name = name ; Address = addr})
```

```
let PersonContactFlowlet =
    Flow.Do {
        let! person = personFlowlet
        let! ct = contactTypeFlowlet
        let! contactDetails = contactFlowlet ct
        return! Flow.Static (finalPage person contactDetails)
    }
    |> Flow.Embed
```

Query expressions

```
query {
    for student in db.Student do
    where (student.ID = 1)
    select student
}
```

The F# 3.0 beta contains a query computation expression with tons of new keywords. How can I define my own keywords in a computation builder?
- from *stackoverflow* -

CustomOperation

The following example shows the extension of the existing `Microsoft.FSharp.Linq.QueryBuilder` class.

F# Copy

```
type Microsoft.FSharp.Linq.QueryBuilder with

    [<CustomOperation("existsNot")>]
    member _.ExistsNot (source: QuerySource<'T, 'Q>, predicate) =
        Enumerable.Any (source.Source, Func<_,_>(predicate)) |> not
```

Syntax

expr :=

    expr { cexpr }

cexpr :=

    let! pat = expr in cexpr

    return expr

    ident arg_1 … arg_n

Custom operator

{| ident arg |}$_{vs,inp}$

$\Rightarrow$ b.CustomOperation (inp@vs, fun vs -> arg)

    where isProjectionParameter(arg)

    and isCustomOperator(ident)

$\Rightarrow$ b.CustomOperation (inp@vs, arg)

    where isCustomOperation(ident)

CustomOperationAttribute

```
[<AttributeUsage(AttributeTargets.Method, AllowMultiple = false)>]
[<Sealed>]
type CustomOperationAttribute =
 class
  new CustomOperationAttribute : string -> CustomOperationAttribute
  member this.AllowIntoPattern : bool with get, set
  member this.IsLikeGroupJoin : bool with get, set
  member this.IsLikeJoin : bool with get, set
  member this.IsLikeZip : bool with get, set
  member this.MaintainsVariableSpace : bool with get, set
  member this.MaintainsVariableSpaceUsingBind : bool with get, set
  member this.Name : string
  member this.IsLikeGroupJoin : bool with get, set
  member this.IsLikeJoin : bool with get, set
  member this.IsLikeZip : bool with get, set
  member this.MaintainsVariableSpace : bool with get, set
  member this.MaintainsVariableSpaceUsingBind  : bool with get, set
 end
```

Usage: CustomOperation

```
type SeqBuilder() =
   member x.For (source : seq<'a>, body : 'a -> seq<'b>) =
    seq { for v in source do yield! body v }
   member x.Yield (item:'a) : seq<'a> = seq { yield item }

  [<CustomOperation("select")>]
   member x.Select (source : seq<'a>, f: 'a -> 'b) : seq<'b> =
     Seq.map f source

let myseq = SeqBuilder()

myseq {
   for i in 1 .. 10 do
   select (fun i -> i + 100)
   }
```

```
val it : seq<int> = seq [101; 102; 103; 104; ...]
```

Usage: ProjectionParameter

```
type SeqBuilder() =
   member x.For (source : seq<'a>, body : 'a -> seq<'b>) =
    seq { for v in source do yield! body v }
   member x.Yield (item:'a) : seq<'a> = seq { yield item }

   [<CustomOperation("select")>]
   member x.Select (source : seq<'a>, [<ProjectionParameter>] f: 'a -> 'b) : seq<'b> =
      Seq.map f source

let myseq = SeqBuilder()
```

```
myseq {
   for i in 1 .. 10 do
   select (i + 100)
   }
```

```
val it : seq<int> = seq [101; 102; 103; 104; ...]
```

{| ident arg |}$_{vs,inp}$ $\Rightarrow$ b.CustomOperation (inp@vs, fun vs -> arg)
where isProjectionParameter(arg) and isCustomOperator(ident)

Usage: MaintainsVariableSpace

```
type SeqBuilder() =
   member x.For (source : seq<'a>, body : 'a -> seq<'b>) =
     seq { for v in source do yield! body v }
   member x.Yield (item:'a) : seq<'a> = seq { yield item }

   ...
   [<CustomOperation("sort", MaintainsVariableSpace = true)>]
   member x.Sort (source : seq<'T>) = Seq.sort source

let myseq = SeqBuilder()

myseq {
   let x = 1
   for i in 1 .. 10 do
   sort
   select (x,i + 100)
   }
```

```
val it : seq<int * int> = seq [(1, 101); (1, 102); ...]
```

Writing custom F# LINQ query builder

- http://tomasp.net/blog/2015/query-translation/

Example: IL DSL (1)

```
> let il = ILBuilder()

// will return 42 when called
> let fortyTwoFn =
    il {
      ldc_i4 6
      ldc_i4_0
      ldc_i4 7
      add
      mul
      ret
    }
val fortyTwoFn : (unit -> int)

> fortyTwoFn ()
val it : int = 42
```

From Keith Battocchi

Example: IL DSL (2)

```
type Stack<'a> = Stack of (ILGenerator -> unit)
type Completed<'a> = Completed of (ILGenerator -> unit)

type ILBuilder() =
  // stack transition: int::int::rest -> int::rest
  [<CustomOperation("add")>]
  member __.Add(Stack f : Stack<int * (int * 'r)>) : Stack<int * 'r> =
    Stack(fun ilg -> f ilg; ilg.Emit(OpCodes.Add))

  // stack transition: int::int::rest -> int::rest
  [<CustomOperation("mul")>]
  member __.Mul(Stack f : Stack<int * (int * 'r)>) : Stack<int * 'r> =
    Stack(fun ilg -> f ilg; ilg.Emit(OpCodes.Mul))

  member __.Run(Completed f : Completed<'a>) : unit -> 'a =
    let dm = DynamicMethod("", typeof<'a>, [||])
    dm.GetILGenerator() |> f
    (dm.CreateDelegate(typeof<System.Func<'a>>) :?> System.Func<'a>).Invoke
```

Method	Typical signature(s)	Description
`Bind`	`M<'T> * ('T -> M<'U>) -> M<'U>`	Called for `let!` and `do!` in computation expressions.
`Delay`	`(unit -> M<'T>) -> M<'T>`	Wraps a computation expression as a function.
`Return`	`'T -> M<'T>`	Called for `return` in computation expressions.
`ReturnFrom`	`M<'T> -> M<'T>`	Called for `return!` in computation expressions.
`Run`	`M<'T> -> M<'T>` or `M<'T> -> 'T`	Executes a computation expression.
`Combine`	`M<'T> * M<'T> -> M<'T>` or `M<unit> * M<'T> -> M<'T>`	Called for sequencing in computation expressions.
`For`	`seq<'T> * ('T -> M<'U>) -> M<'U>` or `seq<'T> * ('T -> M<'U>) -> seq<M<'U>>`	Called for `for...do` expressions in computation expressions.

`TryFinally`	`M<'T> * (unit -> unit) -> M<'T>`	Called for `try...finally` expressions in computation expressions.
`TryWith`	`M<'T> * (exn -> M<'T>) -> M<'T>`	Called for `try...with` expressions in computation expressions.
`Using`	`'T * ('T -> M<'U>) -> M<'U> when 'U :> IDisposable`	Called for `use` bindings in computation expressions.
`While`	`(unit -> bool) * M<'T> -> M<'T>`	Called for `while...do` expressions in computation expressions.
`Yield`	`'T -> M<'T>`	Called for `yield` expressions in computation expressions.
`YieldFrom`	`M<'T> -> M<'T>`	Called for `yield!` expressions in computation expressions.
`Zero`	`unit -> M<'T>`	Called for empty `else` branches of `if...then` expressions in computation expressions.

Acknowledgements

- Some slides from:

https://files.meetup.com/2922732/Computation%20Expressions%20%28April%29.pdf