Computation Expressions

The F# Team
Wonseok Chae

F# Seattle Meetup (4/25)
The basic techniques for functional programming

F# BASIC
Higher-order functions

> let applyTwice f = f <<< f
val applyTwice : ('a -> 'a) -> ('a -> 'a)

> applyTwice ((*) 2) 5;;
val it : int = 20

(*) : (int -> int -> int)
Example: Y Combinator

> let rec y f x = f (y f) x
val y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b

> let fact fact = \_n\_ -> n * fact (n-1)
val fact : (int -> int) -> int -> int

> y fact 5;;
val it : int = 120
Example: Y Combinator

> let rec y f x = f (y f) x
val y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b

> let fact fact = function 0 -> 1 | n -> n * fact (n-1)
val fact : (int -> int) -> int -> int

> y (fact >> fun f n -> printf "%d\n" n; f n) 3;;
3
2
1
0
val it : int = 6
Example: Time Combinator

> let time f x =
  let start = System.DateTime.Now
  try
    f x
  finally
    printf "took %A\n" (System.DateTime.Now - start)
val time : ('a -> 'b) -> 'a -> 'b

> time (y fact) 5;;
took 00:00:00.0029214
val it : int = 120
Continuations (1)

> type Expr = Num of int | Add of Expr * Expr

> let rec eval expr = match expr with
  | Num v -> v
  | Add (e1, e2) -> eval e1 + eval e2

val eval : Expr -> int
Continuations (2)

> type Expr = Num of int | Add of Expr * Expr

> let rec evalC expr cont = match expr with
  | Num v -> cont v
  | Add (e1, e2) ->
    evalC e1 (fun v1 ->
    evalC e2 (fun v2 ->
    cont (v1 + v2)))
val evalC : Expr -> (int -> 'a) -> 'a

> let eval expr = evalC expr (fun x -> x)
val eval : Expr -> int
Classes

> type Evaluator () =
    private member this.EvalC e cont = match e with
        | Num v -> cont v
        | Add (e1, e2) ->
            EvalC (e1, fun v1 ->
                EvalC (e2, fun v2 ->
                    cont (v1 + v2))))
    member this.Eval e = this.EvalC e (fun x -> x)

> let e = new Evaluator()
> e.Eval (Add(Num 1, Num 2))
val it : int = 3
Computation Expressions to “Build Semantically Rich F# Programs”

EXAMPLES
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)
    
}
Asynchronous workflows

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

Asynchronous "non-blocking" action

"Thread" continuation/Event callback
Mechanisms for embedding computational languages

THE GENERAL MECHANISM
Syntax

\[\text{expr} := \]
\[\text{expr} \{ \text{cexpr} \} \]

\[\text{cexpr} := \]
\[\text{let! pat} = \text{expr in cexpr} \]
\[\text{do! expr in cexpr} \]
\[\text{use! pat} = \text{expr in cexpr} \]
\[\text{yield! expr} \]
\[\text{yield expr} \]
\[\text{return! expr} \]
\[\text{return expr} \]
\[\text{for pat in expr do expr} \]

- Computation expression
- Binding computation
- Sequential computation
- Auto clean-up computation
- Yield computation
- Yield result
- Return computation
- Return result
- Yield result
De-sugaring

\[
\text{expr \{ cexpr \}} \quad \Rightarrow \quad \text{let } b = \text{expr} \\
\quad \quad \quad \quad \quad \quad \text{b.Run (b.Delay (fun () -> \{ | cexpr | \}_c))}
\]

\[
\{ | \text{let! pat = expr in cexpr |} \}_c \quad \Rightarrow \quad b.\text{Bind} (\text{expr}, (\text{fun pat -> \{ | cexpr | \}_c})) \\
\{ | \text{do! expr in cexpr |} \}_c \quad \Rightarrow \quad b.\text{Bind} (\text{expr}, (\text{fun () -> \{ | cexpr | \}_c})) \\
\{ | \text{do expr in cexpr |} \}_c \quad \Rightarrow \quad \text{expr; \{ | cexpr | \}_c} \\
\{ | \text{yield! expr |} \}_c \quad \Rightarrow \quad b.\text{YieldFrom} (\text{expr}) \\
\{ | \text{yield expr |} \}_c \quad \Rightarrow \quad b.\text{Yield} (\text{expr}) \\
\{ | \text{return! expr |} \}_c \quad \Rightarrow \quad b.\text{ReturnFrom} (\text{expr}) \\
\{ | \text{return expr |} \}_c \quad \Rightarrow \quad b.\text{Return} (\text{expr}) \\
\{ | \text{for pat in expr do expr |} \}_c \quad \Rightarrow \quad b.\text{For} (\{ | expr | \}_E, (\text{fun pat -> \{ | expr | \}_c)))
\]
De-sugaring

\[
\text{expr } \{ \text{cexpr} \} \quad \Rightarrow \quad \text{let } b = \text{expr} \\
\phantom{\text{expr } \{ \text{cexpr} \}} \phantom{\Rightarrow} \phantom{\text{let } b = \text{expr}} \\
\phantom{\Rightarrow} b.\text{Run} (b.\text{Delay} (\text{fun} () \rightarrow \{ | \text{cexpr} | \}_c))
\]

\[
\{ | \text{let! pat = expr in cexpr |} \}_c \quad \Rightarrow \quad b.\text{Bind} (\text{expr}, (\text{fun pat} \rightarrow \{ | \text{cexpr} | \}_c))
\]

\[
\{ | \text{return expr |} \}_c \quad \Rightarrow \quad b.\text{Return}(\text{expr})
\]

**example**

```plaintext
async \{ \\
    \text{let! x = AsyncRead} () \Rightarrow b.\text{Delay} (\text{fun} () \rightarrow \{ | \text{cexpr} | \}_c) \\
    \text{return x}
\}
```

```plaintext
let b = async \\
\phantom{\text{let b = async}} \phantom{\Rightarrow} \\
\phantom{\Rightarrow} b.\text{Delay} (\text{fun} () \rightarrow \{ | \text{cexpr} | \}_c) \\
\phantom{\Rightarrow} b.\text{Bind(AsyncRead(), fun x -> b.\text{Return x})}
```

'{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 () ⇒ b.Delay (fun () ->
    b.Bind(AsyncRead(), fun x ->
      b.Return x))
  return x
}

let b = async

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**: \( M\langle a\rangle \to (a \to M\langle b\rangle) \to M\langle b\rangle \)
- **Return**: \( a \to M\langle a\rangle \)
Customize how various constructs get executed

CUSTOM WORKFLOWS
Example 1: The Logger Workflow

```ml
let! a = y fact 10
let! b = y fact 10
return a + b
```

type LoggerBuilder () =
    let mutable lastTime = None
    member this.Return e =
        lastTime <- None e
    member this.Bind (value, cont) =
        match lastTime with
        | None -> printfn "took at let! = 0 sec"
        | Some (last : System.DateTime) ->
            let time = System.DateTime.Now - last time.TotalSeconds
            |> printfn "took at let! = %f sec"
            lastTime <- Some System.DateTime.Now
            cont value
    member this.Delay (f) =
        lastTime <- Some System.DateTime.Now
        f()

let logger = new LoggerBuilder()
```
Example 2: The Rounding Workflow

```fsharp
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)
```
Example 3: M-Brace

Developed in F#, it enables the distribution of F# computations to the cloud via their very cool cloud monad.

From http://m-brace.net
Example 4: F# Web Tools

// client invokes server-side code
member x.buttonClicked() =
    client {
        do! serverExecute (x.updatePage())
    }

// server-side code updates components
member x.updatePage() =
    server {
        let data = Database.LoadData()
        do! this.list.set_Data(data)
        do! this.calendar.set_Highlighted(hl)
    }

The modal distinctions between client and server are checked through the use of F# workflows, and LINQ can be used for database access.
Example 5: Formlet *in* WebSharper

```javascript
let ContactFormlet () =
  let contactTypeF =
    Controls.Select 0 [("Phone", ContactType.Phone);
                        ("Address", ContactType.Address)]
  Formlet.Do {
    let! contactType = contactTypeF
    return!
    match contactType with
    | ContactType.Address ->
      Formlet.Map Contact.Address (AddressFormlet ())
    | ContactType.Phone ->
      Input ""
      |> Validator.IsNotEmpty "Enter a valid phone number"
      |> Enhance.WithValidationIcon
      |> Enhance.WithTextLabel "Phone"
      |> Formlet.Map Contact.Phone
  }
```

From [http://websharper.com](http://websharper.com)
Better supports Domain Specific Languages

F# 3.0
Query expressions

```fsharp
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 -
Syntax

expr :=
    expr { cexpr }

cexpr :=
    let! pat = expr in cexpr
    return expr
    ident arg₁ ... argₙ

{[ ident arg |]_{vs,inp}}
    ⇒ b.CustomOperation (inp@vs, fun vs -> arg)
    where isProjectionParameter(arg)
    and isCustomOperator(ident)
    ⇒ 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

```ml
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 }

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

```
let myseq = SeqBuilder()

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

[<AttributeUsage(AttributeTargets.Parameter, AllowMultiple = false)>]
[<Sealed>]
type ProjectionParameterAttribute =
    class
        new ProjectionParameterAttribute : unit -> ProjectionParameterAttribute
    end
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} => 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); ...]
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
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
Example: Slider DSL

• Motivated by *SlideShow: Functional Presentations*

```plaintext
page {
    title "Example: Slider DSL"
    item "Motivated by SlideShow: ...
    code @"...
    button (...
} |> SlideShow.show

slide {
    page1
    page2
} |> SlideShow.show
```
Embedded Domain Specific Languages through computation expressions

DISCUSSION
Language oriented programming

Discriminate Unions (or datatypes)
Computation Expressions
Quotation Expressions

From Jim Weirich’s Speaking the Lingo

Fsslex / fsyacc

VS Studio DSL