Computation expressions and monads

Peter Sestoft

BSWU BFNP 2016-03-15

Agenda

- Computation expressions, or monads
- Sequence expressions as computation exprs.
- Relation to list and array expressions
- Monad laws
- Simple expressions and evaluators
  - Return int, standard evaluation
  - Return int option, evaluation may fail
  - Return int Set, evaluation may produce a set
  - Return int trace, evaluation traces operators
  - Express all this uniformly using comp. expr.

- Thursday: async as computation expression

What is a computation expression?

- A *computation expression* such as

```
seq { for x in [1 .. 3] do yield x*x }
```

  is *syntactic sugar* for a standard functional expression, such as

```
Seq.collect (fun x -> Seq.singleton(x*x))
            [1 .. 3]
```

- This gives a systematic way to combine operations or propagate "background" data
- Computation expressions are sometimes called *monads* (in mathematics, Haskell, ...)

Computation expressions in F#

- seq {...} is a computation expression
- async {...} is a comp. expr. (next week)
- We can define our own computation exprs:
  - Optional result (None or Some)
  - Set of results
  - Trace of operations
  - ...

Sequences as computation expressions (H&R p 281)

- This sequence expression

```
seq { for i in [1 .. 3] do
        for ch in ['a' .. 'd'] do
          yield (i,ch) }
```

- is syntactic sugar for this expression

```
Seq.collect
 (fun i ->
   Seq.collect
    (fun ch ->
      Seq.singleton (i, ch))
    ['a' .. 'd'])
 [1 .. 3]
```

Transformation of seq {...}

The compiler rewrites **`for`** and **`yield`** keywords to normal function calls:

Seq construct C	Transformation T(C)
`for x in e do ce`	`For(e, fun x -> T(ce))`
`yield e`	`Yield(e)`

The **`For`** and **`Yield`** functions must be defined:

```
For   : seq<'a> * ('a -> seq<'b>) -> seq<'b>
Yield : 'a -> seq<'a>
```

```
For(xs, f) = Seq.collect f xs
Yield a    = Seq.singleton a
```

How to define our own mySeq {...}?

- Define a MySeqBuilder class with For, Yield:

```
type MySeqBuilder() =
  member this.For(xs, f) = Seq.collect f xs
  member this.Yield x = Seq.singleton x
```

- Make an object of that class:

```
let mySeq = new MySeqBuilder()
```

- The object can now indicate a comp expr:

```
mySeq { for i in [1 .. 3] do
          for ch in ['a' .. 'd'] do
            yield (i, ch) };;
```

Homemade sequence expression

Transformation and implementation

- The computation expression

```
mySeq { for x in [1..3] do yield x*x }
```

  gets transformed (by the compiler) to

```
mySeq.For([1..3], fun x -> mySeq.Yield(x*x))
```

- Using the definitions of functions For and Yield in mySeq, this means

```
Seq.collect (fun x -> Seq.singleton (x*x))
            [1..3]
```

Understanding the H&R example

```
seq { for i in [1 .. 3] do
        for ch in ['a' .. 'd'] do
          yield (i,ch) }
```

- Lift inner **for** out as a function:

```
let inner i =
  seq { for ch in ['a' .. 'd'] do yield (i,ch) }
```

Same thing

```
let inner i =
  Seq.collect (fun ch -> Seq.singleton (i, ch)) ['a' .. 'd']
```

- Outer **for** is just this:

```
Seq.collect (fun i -> inner i) [1 .. 3]
```

- So in total

```
Seq.collect (fun i ->
  Seq.collect (fun ch -> Seq.singleton (i, ch)) ['a' .. 'd'])
  [1 .. 3]
```

List and array expressions

- The F# list expression:

```
[ for x in [1..3] do yield x*x ]
```

- is syntactic sugar for the seq expression

```
Seq.toList(seq {for x in [1..3] do yield x*x })
```

- Similarly for F# array expressions:

```
[| for x in [1..3] do yield x*x |]
```

- See F# Specification §6.3.13 and §6.3.14
- F# has no "list computation expression", it boils down to seq computation expressions

Questions

- In seq {...} expressions one can use "**`if`**"

```
let sift a xs =
  mySeq { for n in xs do
             if n % a <> 0 then
               yield n };;
```

- Q1: What function to add to MySeqBuilder to support the **`if`** operator? (H&R Table 12.2)

- Q2: What member of the Seq module should be used to define it? (H&R Table 11.1)

For=Bind=let!, and Yield=Return

- For and Yield are special names that make sense in seq{...} expressions
- Normal names are Bind/let! and Return
- One could define sillySeq using Bind/Return:

```
type SillySeqBuilder() =
  member this.Bind(xs, f) = Seq.collect f xs
  member this.Return x = Seq.singleton x

let sillySeq = new SillySeqBuilder()

sillySeq {
  let! i = [1 .. 3]
  let! ch = ['a' .. 'd']
  return (i, ch) }
```

```
seq {
  for i in [1 .. 3] do
    for ch in ['a' .. 'd'] do
      yield (i,ch) }
```

More sensible, but exact same meaning

Kært barn har mange navne

- Function **`Bind`** in computation expression is
  - **`for`** and **`For`** in **`seq {...}`**
  - **`List.collect`** on F# lists
  - **`Seq.collect`** on F# sequences
  - **`flatMap`** on Scala and Haskell lists, sequences, ...
  - **`flatMap`** in Java 8 streams
  - **`SelectMany`** in Microsoft Linq (eg. C#P p. 205)
  - **`bind`** in monads

- Function Return in computation expressions
  - **`yield`** in **`seq {...}`**
  - **`(fun x -> [x])`** on F# lists
  - **`Seq.singleton`** on F# sequences
  - **`unit`** or **`return`** in monads

Transformation of computation expressions

The compiler rewrites to normal function calls:

Comp. exp. construct C	Transformation T(C)
`let! x = e` `ce`	`Bind(e, fun x -> T(ce))`
`return e`	`Return(e)`
`return! e`	`ReturnFrom(e)`

The functions must have suitable types:

```
Bind    : com<'a> * ('a -> com<'b>) -> com<'b>
Return  : 'a -> com<'a>
ReturnFrom : com<'a> -> com<'a>
```

See H&R Tables 12.2 and 12.3 for more

The option {...} computation expr. A form of error propagation

```
optM { let x = 56
       let! y = Some 78
       return x+y };;
```

```
let x = 56
optM.Bind(Some 78,
  fun y ->
  optM.Return(x+y))
```

Some 134

```
optM { let x = 56
       let! y = Some 78
       let! z = None
       return x+y };;
```

```
let x = 56
optM.Bind(Some 78,
  fun y ->
  optM.Bind(None,
    fun z ->
    optM.Return(x+y)))
```

None

```
type OptionBuilder() =
    member this.Bind(x, f) =
        match x with
        | None   -> None
        | Some v -> f v
    member this.Return(x) = Some x
let optM = OptionBuilder()
```

let!

return

Question: What's happening here

```
type OptionBuilder() =
    member this.Bind(x, f) =
        printfn "this.Bind: %A" x
        match x with
        | None    -> None
        | Some v -> f v
    member this.Return(x) = Some x
let optM = OptionBuilder()
```

Print when called

```
optM { let x = 56
       let! y = Some 78
       let! z = None
       let! v = Some 42
       return x+y+v };;
```

New line here

Monad laws

- For(Yield a, f) = f(a)

  collect f (singleton a) = f(a)
- For(xs, Yield) = xs

  collect singleton xs = xs

- The laws are the same with
  - Bind instead of For, and Return instead of Yield
- Let's check them for the **`optM`** monad

A standard simple evaluator

- Very simple expressions like 7 + 9 * 10

```
type expr =
   | CstI of int
   | Prim of string * expr * expr
```

```
Prim("+",
   CstI(7),
   Prim("*", CstI(9),
```

- A simple evaluator:

```
let rec eval1 e : int =
      match e with
      | CstI i -> i
      | Prim(op, e1, e2) ->
          let v1 = eval1 e1
          let v2 = eval1 e2
          match op with
          | "+" -> v1 + v2
          | "*" -> v1 * v2
          | "/" -> v1 / v2
```

```
let opEval op v1 v2 : int =
      match op with
      | "+" -> v1 + v2
      | "*" -> v1 * v2
      | "/" -> v1 / v2
```

```
let rec eval2 e : int =
      match e with
      | CstI i -> i
      | Prim(op, e1, e2) ->
          let v1 = eval2 e1
          let v2 = eval2 e2
          opEval op v1 v2
```

An evaluator that may fail (w. None)

```
let opEvalOpt op v1 v2 : int option =
    match op with
    | "+" -> Some(v1 + v2)
    | "*" -> Some(v1 * v2)
    | "/" -> if v2 = 0 then None else Some(v1 / v2)
```

```
let rec optionEval2 e : int option =
    match e with
    | CstI i -> Some i
    | Prim(op, e1, e2) ->
        match optionEval2 e1 with
        | None -> None
        | Some v1 ->
            match optionEval2 e2 with
            | None -> None
            | Some v2 -> opEvalOpt op v1 v2
```

An evaluator giving a set of results

```
let opEvalSet op v1 v2 : int Set =
    match op with
    | "+" -> Set [v1 + v2]
    | "*" -> Set [v1 * v2]
    | "/" -> if v2 = 0 then Set.empty else Set [v1 / v2]
    | "choose" -> Set [v1; v2]
```

```
let rec setEval1 e : int Set =
    match e with
    | CstI i -> Set [i]
    | Prim(op, e1, e2) ->
    let s1 = setEval1 e1
    let yss =
       Set.map (fun v1 ->
                let s2 = setEval1 e2
                let xss = Set.map (fun v2->opEvalSet op v1 v2) s2
                Set.unionMany xss)
                s1
    Set.unionMany yss
```

An evaluator tracing the operators

```
type 'a trace = string list * 'a

let opEvalTrace op v1 v2 : int trace =
    match op with
    | "+" -> (["+"], v1 + v2)
    | "*" -> (["*"], v1 * v2)
    | "/" -> (["/"], v1 / v2)
```

```
let rec traceEval1 e : int trace =
    match e with
    | CstI i -> ([], i)
    | Prim(op, e1, e2) ->
        let (trace1, v1) = traceEval1 e1
        let (trace2, v2) = traceEval1 e2
        let (trace3, res) = opEvalTrace op v1 v2
        (trace1 @ trace2 @ trace3, res)
```

A mess; comp expr to the rescue

- The four evaluators look very different
- ... and very complicated

- By defining the combining operations as computation expressions,
  - the evaluators all get to look the same
  - the evaluators look much simpler

An evaluator that may fail, NEW

```
type OptionBuilder() =
    member this.Bind(x, f) =
        match x with
        | None   -> None
        | Some v -> f v
    member this.Return x = Some x
    member this.ReturnFrom x = x

let optionM = OptionBuilder();;
```

```
let rec optionEval3 e : int option =
    match e with
    | CstI i -> optionM { return i }
    | Prim(op, e1, e2) ->
        optionM { let! v1 = optionEval3 e1
                  let! v2 = optionEval3 e2
                  return! opEvalOpt op v1 v2 }
```

An evaluator ... set of results, NEW

```
type SetBuilder() =
    member this.Bind(x, f) =
        Set.unionMany (Set.map f x)
    member this.Return x = Set [x]
    member this.ReturnFrom x = x

let setM = SetBuilder();;
```

```
let rec setEval3 e    : int Set =
    match e with
    | CstI i -> setM    { return i }
    | Prim(op, e1, e2) ->
        setM    { let! v1 = setEval3 e1
                  let! v2 = setEval3 e2
                  return! opEvalSet op v1 v2 }
```

An evaluator ... trace operators, NEW

```
type TraceBuilder() =
    member this.Bind(x, f) =
        let (trace1, v) = x
        let (trace2, res) = f v
        (trace1 @ trace2, res)
    member this.Return x = ([], x)
    member this.ReturnFrom x = x

let traceM = TraceBuilder();;
```

```
let rec traceEval3 e  : int trace =
    match e with
    | CstI i -> traceM  { return i }
    | Prim(op, e1, e2) ->
        traceM  { let! v1 = traceEval3 e1
                  let! v2 = traceEval3 e2
                  return! opEvalTrace op v1 v2 }
```

A standard evaluator, NEW

```
type IdentityBuilder() =
    member this.Bind(x, f) = f x
    member this.Return x = x
    member this.ReturnFrom x = x

let identM = new IdentityBuilder();;
```

```
let rec eval3 e        : int =
    match e with
    | CstI i -> identM  { return i }
    | Prim(op, e1, e2) ->
        identM  { let! v1 = eval3 e1
                  let! v2 = eval3 e2
                  return! opEval op v1 v2 }
```

Reflections on computation expressions

- They reveal similarities
  - between different kinds of computations
  - between different kinds of data: list, seq, option, ...
- They clarify the structure of the evaluators

- Unfortunately, in F# a computation expression builder (optionM, setM, traceM, identM) cannot be a parameter to a function
- Hence one cannot have a single "super-eval" that unifies all the {opt,set,trace}Evals
- ... but in Scala one can!

Types of the monad functions

Monad	Bind/For	Return/Yield
mySeq	`seq<'a> * ('a -> seq<'c>) -> seq<'c>`	`'a -> seq<'a>`
optionM	`'a option * ('a -> 'b option) -> 'b option`	`'a -> 'a option`
setM	`Set<'a> * ('a -> Set<'b>) -> Set<'b>`	`'a -> Set<'a>`
traceM	`('a list * 'b) * ('b -> 'a list * 'c) -> 'a list * 'c`	`'a -> 'b list * 'a`
traceM	`'b trace * ('b -> 'c trace) -> 'c trace`	`'a -> 'a trace`
monad M	`'a M * ('a -> 'b M) -> 'b M`	`'a -> 'a M`

References

- F# computation expressions
  - Hansen and Rischel chapter 12
  - http://en.wikibooks.org/wiki/F_Sharp_Programming/Computation_Expressions
  - http://msdn.microsoft.com/en-us/library/dd233182.aspx