Lec 15: Computation Expression

CS220: Programming Principles

Sang Kil Cha
In-Class Activity #17

In this activity, you will write a parallel word counter function using Async. Fill in the `map` function.
Computation Expression
Recall Asynchronous Computations

async { exprs ... } was an example of computation expressions. Computation expressions provide a convenient syntax for writing computations.
Another Example: seq

The seq computation expression has a similar form: `seq { ... }`. And it helps build sequence expressions.

```plaintext
seq { for i = 0 to 5 do yield (i, i * i) }
seq { while true do yield 1 } // Infinite sequence.
```

Compared to Seq.unfold, which one is easier to understand?
What is Similar?

Given the above examples of computation expressions, i.e., `seq` and `async`, is there any common thing that you can figure out?

They return a “wrapped” type\(^1\), called `computation`.

\(^1\)They are wrapped by a type constructor.
Monads

A monad is a design pattern that allows structuring programs generically while automating away boilerplate code needed by the program logic\(^2\).

Monads allow us to hide some low-level details of computations.

\(^2\)https://en.wikipedia.org/wiki/Monad_(functional_programming)
What Were Hidden in `async` and `seq`?

- When building an `async` computation, all the low-level thread management code was hidden from the expression.
- When building a `seq` computation, the details about constructing cons cells and maintaining its state were hidden.
Monads in this Class

A monad is a triple of a generic type constructor \((M<'T>)\) and the two following operations.

\[
\text{val } \text{Bind}: \ M<'T> \times ('T \rightarrow M<'U>) \rightarrow M<'U>
\]

\[
\text{val } \text{Return}: \ 'T \rightarrow M<'T>
\]

✔ Monads are like a box: it is a box with a value of type \('T\), and the function \(('T \rightarrow M<'U>)\) takes the value from the box and returns another box of type \(M<'U>\).
Motivating Example

```ml
let inc x = x + 1
let dec x = x - 1
// Concise and easy to understand
let id = inc >> dec
```

Function composition is elegant and easy to understand.
Motivating Example (cont’d)

```typescript
type ResultWithDebugMessage<'a'> = {
    Result: 'a
    DbgMsg: string
}
let inc x = { Result = x + 1; DbgMsg = "incremented" }
let dec x = { Result = x - 1; DbgMsg = "decremented" }
let id = inc >> dec // type mismatch.
```

Works, but not elegant.

```typescript
let id x =
    let r1 = inc x
    let r2 = dec r1.Result
    { Result = r2.Result; DbgMsg = r1.DbgMsg + "\n" + r2.DbgMsg }
```
Make it Combinable

```ml
let combine f r =
  let r' = f r.Result
  { r' with DbgMsg = r.DbgMsg + "\n" + r'.DbgMsg }

let id = inc >> combine dec
```

The `id` function combines `inc` and `dec` in an elegant manner.
Return the Boxed Type

We want to make `ResultWithDebugMessage` value from an integer: `wrap` function simply wraps a value without any debugging message.

```fsharp
let wrap r = { Result = r; DbgMsg = "" }
let id =
  inc
  >> combine dec
  >> combine (fun x -> x + 1 |> wrap)
  >> combine dec
```
Signatures of `combine` and `wrap`:

val combine: ('a -> ResultWithDebugMessage<'b>) -> ResultWithDebugMessage<'a> -> ResultWithDebugMessage<'b>

val wrap: 'a -> ResultWithDebugMessage<'a>

val Bind: M<'T> * ('T -> M<'U>) -> M<'U>

(Same as `combine` after swapping the argument order)

val Return: 'T -> M<'T>

ResultWithDebugMessage<'a> was a monad!
We can logically combine functions while hiding some details behind the scene with monads. Typically we define a bind operator ( >>= ), which is an infix-operator for the Bind function discussed above.

```ocaml
let (>>=) m f = combine f m
let id x = inc x >>= dec >>= inc >>= dec
```
Enhancing The Expressiveness
Let-Bindings Revisited

We can always convert let-bindings to a function with nested function calls: function calls another function, and the function calls another function, and so on.

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

```
1 |> fun x ->
 2 + x |> fun y ->
  x * y |> fun z ->
       z
```
Creating a Bind Function

Let us now create a `bind` function that takes in a value and a function, and apply the value to the function (as in the pipe operator).

```ml
let bind x f = f x
let ret x = x
bind 1 (fun x ->
    bind (2 + x) (fun y ->
        bind (x * y) (fun z ->
            ret z)))
```

This is so-called “continuation passing style” (CPS).
Continuation Passing Style?

In CPS, functions always end with a function that we call \textit{continuation}, which describes what to do next.

\begin{verbatim}
// Normal
let add a b = a + b

// CPS
let add a b cont = cont (a + b)
\end{verbatim}

\textbf{★ Food for Thought.} CPS forces us to write tail-recursive functions, but it doesn’t mean that it helps reduce memory consumption.
CPS Bindings vs. Let-Bindings

Syntactically different but semantically the same.

```
bind 1 (fun x ->
    bind (2 + x) (fun y ->
        bind (x * y) (fun z ->
            ret z)))

let x = 1
let y = 2 + x
let z = x * y
z
```
Hiding Complex Logic

If we can transform the chain of `bind-ret` function calls into expressions that look like `let-bindings`, and if our language supports such a transformation, then we can hide some complex logic under a beautiful language (with `let!` and etc.).

```ocaml
let bind x f =
    printfn "you can do some complex things here."
    f x

bind 1 (fun x ->
    bind (2 + x) (fun y ->
        bind (x * y) (fun z ->
            ret z)))
```
Example: Safe Division

Safe division function.

```ml
let safeDiv a b =
  if b = 0 then None
  else Some (a / b)
```
Too Many Nested Checks

```plaintext
// unsafe div
let x = (((a / b) / c) / d) / e
// safe div
let x' =
    match safeDiv a b with
    | None -> None
    | Some r ->
        match safeDiv r c with
        | None -> None
        | Some r ->
            match safeDiv r d with
            | None -> None
            | Some r -> safeDiv r e
```
Observation

The nested match expressions follow a CPS. We can write our own “bind” function to connect them!

```ocaml
let bind (x, f) =
    match x with
    | None -> None
    | Some m -> f m

let ret x = Some x

bind (safeDiv a b, fun r ->
    bind (safeDiv r c, fun r ->
        bind (safeDiv r d, fun r ->
            bind (safeDiv r e, fun r ->
                ret r)))))
```
Built-in Binder: Option.bind

Works the same, but it takes a continuation first.

val Option.bind: ('a -> 'b option) -> 'a option -> 'b option

safeDiv a b |> Option.bind (fun r ->
safeDiv r c |> Option.bind (fun r ->
safeDiv r d |> Option.bind (fun r ->
safeDiv r e |> Option.bind (fun r ->
ret r))))
A computation expression builder is a class that contains several member functions such as `Bind` and `Return`.

```plaintext
val __.Bind:  M<'T>  *  ('T -> M<'U>)  ->  M<'U>
val __.Return:  'T  ->  M<'T>
```

We can define our own builder to create our computation expressions.
Maybe Computation Expression

```fsharp
type MaybeBuilder () =
    member __.Bind (m, f) = Option.bind f m
    member __.Return (m) = Some m

let maybe = MaybeBuilder ()
```
maybe {
  let! r = safeDiv a b
  let! r = safeDiv r c
  let! r = safeDiv r d
  let! r = safeDiv r e
  return r
}

• The Bind member corresponds to the let! expression.
• The Return member corresponds to the return expression.
DB Example

match Email.create emailInput with
| Some email ->
  match Name.create nameInput with
  | Some name ->
    match ID.create idInput with
    | Some id ->
      DB.insert db email name id // insert into the DB
    | None -> db // return the DB as it is
  | None -> db
| None -> db
maybe {
  let! email = Email.create emailInput
  let! name = Name.create nameInput
  let! id = ID.create idInput
  return DB.insert email name id
}
Conclusion
Further Readings

• https://fsharpforfunandprofit.com/series/computation-expressions.html
Question?