Programovací jazyky F# a OCaml

Chapter 6.

Sequence expressions and
computation expressions (*aka* monads)

Sequence expressions 1.
(generating sequences)

Sequence expressions

» Lazily generated sequenc

```
> let nums = seq {
    let n = 10
    yield n + 1
    printfn "second.."
    yield n + 2 };;
val nums : seq<int>

> nums |> List.ofSeq;;
second..
val it : int list = [11; 12]

> nums |> Seq.take 1 |> List.ofSeq;;
val it : int list = [11]
```

seq identifier specifies that we're writing sequence expression

We can use all standard F#

Generates next element

Nothing runs yet!

Calculate all elements

Move to first **yield** only

Sequence expressions

» Sequences can be composed using **yield!**

```
let capitals = [ "London"; "Prague" ]

let withNew(x) =
   seq { yield x
         yield "New " + x }

let allCities =
  seq { yield "Seattle"
        yield! capitals
        yield! withNew("York") }
```

Standard list of values

Function that generates sequence with two elements

Yield all elements of the list

Yield both generated names

Generating complex sequences

» Thanks to **yield!** we can use recursion

```
let rec range(nfrom, nto) = seq {
  if (nfrom < nto) then
    yield nfrom
    yield! range(nfrom + 1, nto) }
```

Recursive function

Terminates sequence if false

Tail-call like situation

Recursive call in tail-cal position is optimized

We can write infinite sequences too…

```
let rec numbers(nfrom) = seq {
  yield nfrom
  yield! numbers(nfrom + 1) }
```

DEMO

Working with sequences using HOFs, working with infinite sequences

Sequence expressions 2.
(processing sequences)

Processing sequences

- Calculate squares of all numbers…

```
let squares = seq {
  for n in numbers(0) do
    yield n * n }
```

Iterate over the source

Generate 0 or more elements to output…

- “Cross join” of two sequences

```
let cities = [ ("New York", "USA"); ("London", "UK");
               ("Cambridge", "UK"); ("Cambridge", "USA") ]
let entered = [ "London"; "Cambridge" ]

seq {
  for name in entered do
    for (n, c) in cities do
      if n = name then yield sprintf "%s from %s" n c }
```

Find 0 or more matching cities

How does it work?

» Each **for** translated to a call to **collect**

First step, replace the outer **for**:

```
entered |> Seq.collect (fun name ->
  seq { for (n, c) in cities do
          if n = name then
            yield sprintf "%s from %s" n c })
```

Second step, replace the inner **for**:

```
entered |> Seq.collect (fun name ->
  cities |> Seq.collect (fun (n, c) ->
    if n = name then [ sprintf "%s from %s" n c ]
    else []))
```

Computation expressions

Introducing monads…

» A type **M<'T>** with two operations:

Bind operation:

```
Option.bind  :
  ('a -> option<'b>) -> option<'a> -> option<'b>
Seq.collect :
  ('a -> seq<'b>)    -> seq<'a>    -> seq<'b>
```

Return operation:

```
Seq.singleton : 'a -> seq<'a>
Option.Some   : 'a -> option<'a>
```

Algebraically : some axioms should be true…

Introducing monads…

» Multiplying elements in sequences

```
seq {
   for n in numbers1 do
      for m in numbers2 do
         yield n * m }
```

» Writing similar computation for options…

```
option {
   for n in tryReadInt() do
      for m in tryReadInt() do
         yield n * m }
```

```
option {
   let! n = tryReadInt()
   let! m = tryReadInt()
   return n * m }
```

Syntax with **for** and **yield** is used with sequences…

Behavior for sample inputs

Values	Input #1	Input #2	Output
Lists	[2; 3]	[10; 100]	[20; 200; 30; 300]
Options	Some(2)	Some(10)	Some(20)
Options	Some(2)	None	None
Options	None	*(not required)*	None

How does it work?

» The code is translated to member calls…

```
option {
   let! n = tryReadInt()
   let! m = tryReadInt()
   return n * m }
```

let! – translated to “Bind” operation

return – translated to “Return” operation

```
option.Bind(tryReadInt(), fun n ->
   option.Bind(tryReadInt(), fun m ->
      let add = n + m
      let sub = n - m
      value.Return(n * m) ))
```

Implementing builder

» Simple object type with two members:

```
type OptionBuilder() =
  member x.Bind(v, f) = v |> Option.bind f
  member x.Return(v) = Some(v)
let option = new OptionBuilder()
```

More about objects in the next lecture :-)

» Members should have the usual types:

```
Bind   : ('a -> M<'b>) -> M<'a> -> M<'b>
Return : 'a -> M<'a>
```

Computation expression for “resumptions”

Motivation

- We want to run a computation step-by-step
  - E.g. someone calls our "tick" operation
  - It should run only for a reasonably long time
- How to write calculations that take longer?
  - We can run one step during each "tick"
  - How to turn usual program into stepwise code?
- **Example:** Visual Studio IDE and IntelliSense

Designing computation expression

- **First step:** We need to define a type representing the computation or the result
  - Computation that may fail: `option<'a>`
  - Computation returning multiple values: `seq<'a>`
- The `Resumption<'a>` type:
  - Either finished or a function that runs next step

```
type Resumption<'a> =
  | NotYet of (unit -> Resumption<'a>)
  | Result of 'a
```

Implementing computation expression

» **Second step:** Defining ‘bind’ and ‘return’:

Return should have a type *'a -> Resumption<'a>*

```
let returnR v = Result(v)
```

Bind is more complicated. The type should be:

Resump<'a> -> ('a -> Resump<'b>) -> Resump<'b>

```
let rec bindR v f =
  NotYet(fun () ->
    match v with
    | NotYet calcV -> bindR (calcV()) f
    | Result value -> f value)
```

‘v’ already took some time to run

Run the next step and then bind again…

We return result as the next step

Return the rest of the computation

Designing computation expression

» **Third step:** Computation builder

```
type ResumptionBuilder() =
  member x.Bind(v, f) = bindR v f
  member x.Return(v) = returnR v
let resumable = new ResumptionBuilder()
```

Builder instance

» Writing code using resumptions:

```
let foo(arg) = resumable {
  return expensiveCalc(arg) }

let bar() = resumable {
  let! a = foo(1)
  let! b = foo(2)
  return a + b }
```

Single-step resumption

Compose steps

Returns **Resumption<int>**

DEMO

Programming with resumptions

Resumptions summary

» Writing code using resumptions

Simple transformation of source code

Add “resumable { }” and **let!** with **return**

» Using resumptions

Step-by-step evaluation of computations

Micro-threading – nonpreemtive parallelism (interleaving execution of multiple resumables)

Asynchronous workflows

Motivation

» Downloading web pages from the internet:

```
open System
open System.Net
open System.IO

let syncDownload(url:string) =
  let req = WebRequest.Create(url)
  let rsp = req.GetResponse()
  use stream = rsp.GetResponseStream()
  use reader = new StreamReader(stream)
  let html = reader.ReadToEnd()
  printfn "%s" html

let urls = [ "http://www.microsoft.com"; ... ]
for url in urls do syncDownload(url)
```

Initialize request

Send request **(can take long time!)**

Download page **(can take long time!)**

Motivation

- Performing slow or unreliable I/O
  - Can take a long time and may fail
  - We don’t want to block the current thread!
- Run operation on a background thread?
  - Threads are expensive (on Windows)
  - We’re just waiting! Not doing anything useful
- The right approach: Asynchronous calls

Asynchronous calls in .NET

» .NET provides BeginXyz methods…

```
let uglyAsyncDownload(url:string) =
  let req = WebRequest.Create(url)
  req.BeginGetResponse((fun ar1 ->
    let rsp = req.EndGetResponse(ar1)
    use stream = rsp.GetResponseStream()
    use reader = new StreamReader(stream)
    let html = reader.ReadToEnd()
    printfn "%s" html
  ), null) |> ignore

for url in urls do uglyAsyncDownload(url)
```

Runs the function when response is received

Oops! There is no **BeginReadToEnd**

Starts all computations

The F# solution…

- Asynchronous workflows
  - Computation that eventually calculates some result and then calls the provided function
- The Async<'a> type (simplified):

```
type Async<'a> =
  | Async of (('a -> unit) -> unit)
```

Returns nothing now.

Takes a continuation ('a -> unit)

When the operation completes (later), invokes the continuation

Asynchronous workflows

» Downloading web pages using workflows

```
#r "FSharp.PowerPack.dll"
open Microsoft.FSharp.Control.WebExtensions

let asyncDownload(url:string) = async {
  let req = WebRequest.Create(url)
  let! rsp = req.AsyncGetResponse()
  use stream = rsp.GetResponseStream()
  use reader = new StreamReader(stream)
  let! html = reader.AsyncReadToEnd()
  printfn "%s" html }

[ for url in urls do yield asyncDownload(url) ]
  |> Async.Parallel |> Async.RunSynchronously
```

Reference necessary libraries & namespaces

asynchronous operation

List of computations to run

Compose & run them

Asynchronous workflows

» We can use standard control structures

Recursion or even imperative F# features

```
let asyncReadToEnd(stream:Stream) = async {
  let ms = new MemoryStream()
  let read = ref -1
  while !read <> 0 do
    let buffer = Array.zeroCreate 1024
    let! count = stream.AsyncRead(buffer, 0, 1024)
    ms.Write(buffer, 0, count)
    read := count
  ms.Seek(0L, SeekOrigin.Begin) |> ignore
  return (new StreamReader(ms)).ReadToEnd() }
```