Lecture 10 CSCI: Option Monad

Sebastiaan Joosten

September 27th

Outline

- Option monad
- Homework description

Pipelining Option-functions

- Let's implement overflow-safe addition and safe integer division

```
let plus x y
 = let s = x + y in
   match (x >= 0, y >= 0, s>=0) with
   | (true, true, true) -> Some s
   | (false, false, false) -> Some s
   | (true, false, _) -> Some s
   | (false, true, _) -> Some s
   | (_, _, _) -> None

let divide x
 = function 0 -> None | y -> Some (x / y)
```

Pipelining Option-functions

- Let's do two more:

```
let double x = plus x x
let plus1 x = plus 1 x
```

- The point of these is not their implementation, it's their types:
- plus : int -> int -> int option
- divide : int -> int -> int option
- double : int -> int option
- plus1 : int -> int option

Pipelining Option-functions

- plus : int -> int -> int option
- divide : int -> int -> int option
- double : int -> int option
- plus1 : int -> int option

- We'd like to string together computations like so:
- 3 |> plus1 |> double = Some 8

- Of course, this doesn't type-check…
- Solution: let's replace |> by a different function!

Pipelining Option-functions

- plus : int -> int -> int option
- divide : int -> int -> int option
- double : int -> int option
- plus1 : int -> int option

- We'd like to string together computations like so:
- 3 |> plus1 >>= double = Some 8
- >>= is pronounced 'bind'
- What's the type of >>=?
- (>>=) : 'type of 3 |> plus1' -> 'type of double' -> 'type of Some 8'

Pipelining Option-functions

- plus : int -> int -> int option
- divide : int -> int -> int option
- double : int -> int option
- plus1 : int -> int option

- We'd like to string together computations like so:
- 3 |> plus1 >>= double = Some 8

- What's the type of >>=?
- (>>=) : 'type of Some 4' -> 'type of double' -> 'type of Some 8'

Pipelining Option-functions

- plus : int -> int -> int option
- divide : int -> int -> int option
- double : int -> int option
- plus1 : int -> int option

- We'd like to string together computations like so:
- 3 |> plus1 >>= double = Some 8

- What's the type of >>=?
- (>>=) : 'type of Some 4' -> (int -> int option) -> 'type of Some 8'

Pipelining Option-functions

- plus : int -> int -> int option
- divide : int -> int -> int option
- double : int -> int option
- plus1 : int -> int option

- We'd like to string together computations like so:
- 3 |> plus1 >>= double = Some 8

- What's the type of >>=?
- (>>=) : int option -> (int -> int option) -> int option

Pipelining Option-functions

- We'd like to string together computations like so:
- 3 |> plus1 >>= double = Some 8

- What's the type of >>=?
- (>>=) : int option -> (int -> int option) -> int option

- let (|>) x f = f x
- let (>>=) x f = …

Pipelining Option-functions

- We'd like to string together computations like so:
- 3 |> plus1 >>= double = Some 8

- What's the type of >>=?
- (>>=) : int option -> (int -> int option) -> int option

- let (|>) x f = f x

```
let (>>=) x f = match x with
    …
```

Pipelining Option-functions

- We'd like to string together computations like so:
- 3 |> plus1 >>= double = Some 8

- What's the type of >>=?
- (>>=) : int option -> (int -> int option) -> int option

- let (|>) x f = f x

```
let (>>=) x f = match x with
   | None -> …
   | Some y ->
```

Pipelining Option-functions

- We’d like to string together computations like so:
- 3 |> plus1 >>= double = Some 8

- What’s the type of >>=?
- (>>=) : int option -> (int -> int option) -> int option

- let (|>) x f = f x

```
let (>>=) x f = match x with
    | None -> None
    | Some y -> …
```

Pipelining Option-functions

- We'd like to string together computations like so:
- 3 |> plus1 >>= double = Some 8

- What's the type of >>=?
- (>>=) : int option -> (int -> int option) -> int option

- let (|>) x f = f x

```
let (>>=) x f = match x with
    | None -> None
    | Some y -> f y
```

Pipelining Option-functions

- We'd like to string together computations like so:
- 3 |> plus1 >>= double = Some 8

```
utop # 3 |> plus1 >>= double;;
- : int option = Some 8
```

- let (|>) x f = f x

```
let (>>=) x f = match x with
    | None -> None
    | Some y -> f y
```

Pipelining Option-functions

- The mixing of |> and >>= is perhaps a bit ugly…
- instead we can write:

```
utop # Some 3 >>= plus1 >>= double;;
- : int option = Some 8
```

- let (|>) x f = f x

```
let (>>=) x f = match x with
    | None -> None
    | Some y -> f y
```

Generalization aside

- The >>= operation in these slides is defined for the int option datatype.
- Actually, if we read >>= better, it's for 'a option:

```
utop # let (>>=) x f = match x with
    | None -> None
    | Some y -> f y;;
val ( >>= ) : 'a option -> ('a -> 'b option) -> 'b option
```

- In real life, people define >>= for a huge set of different data-types. We will only look at the 'a option one here.
- However, we'll try to do everything option related with just '>>=', so no more "match" statements after this.
- Reason: this way, this lecture applies to *any* >>= definition.

Pipelining Option-functions

- We can make these as long as we like…

```
utop # Some 3 >>= plus1 >>= double;;
- : int option = Some 8
```

- Where |> was free
  >>= actually does a (very simple) computation for us

… functions with two arguments

- Let’s calculate: 4 + 6 = 10
- 6 |> (4 |> (+))
- Now with our function…
- 6 |> (4 |> plus) = Some 10
- … but what if 6 and 4 are the result of another computation? (i.e. ‘Some 6 / Some 4’)
- Some 6 >>= (4 |> plus) = Some 10 …
- almost there… what can we write for ‘4 |> plus’?

… functions with two arguments

```
utop # Some 4 |> plus;;
Error: This expression has type int -> int -> int option
       but an expression was expected of type int option -> 'a
       Type int is not compatible with type int option
```

… functions with two arguments

```
utop # Some 4 |> plus;;
Error: This expression has type int -> int -> int option
       but an expression was expected of type int option -> 'a
       Type int is not compatible with type int option

utop # Some 4 >>= plus;;
Error: This expression has type int -> int -> int option
       but an expression was expected of type int -> 'a option
       Type int -> int option is not compatible with type
         'a option
```

… functions with 2 arguments

- Let’s apply the currying idea!
- Some 4 >>= (fun nr_four -> Some 6 >>= plus nr_four)

… functions with 2 arguments

- Let’s apply the currying idea!

- Some 4 >>= (fun nr_four -> Some 6 >>= plus nr_four)

- this part: (fun nr_four -> Some 6 >>= plus nr_four)
- it’s a function that returns an ‘int option’
- internally, it applies ‘Some 6’ to ‘plus nr_four’
- here, ‘plus nr_four’ is also a function that returns an ‘int option’

- this is writing 4 + 6 (in that order)

… functions with 2 arguments

- Let’s apply the currying idea!

- Some 4 >>= (fun nr_four -> Some 6 >>= plus nr_four)

- (fun nr_four -> Some 6 >>= plus nr_four)
- should read like: 6 |> (+)
- or: (6 + …)

- I’ll admit: it doesn’t read nicely

Towards nicer notations

- If (Some 6 >>= plus) doesn't type-check,
  … and should really read:
  (fun nr_four -> Some 6 >>= plus nr_four)
- can't we define yet another >>=-like function?

Towards nicer notations

- If (Some 6 >>= plus) doesn't type-check,
  … and should really read:
  (fun nr_four -> Some 6 >>= plus nr_four)
- can't we define yet another >>=-like function?
- let (>>==) x f = …

Towards nicer notations

- If (Some 6 >>= plus) doesn’t type-check,
  … and should really read:
  (fun nr_four -> Some 6 >>= plus nr_four)
- … should we define yet another >>=-like function?
- let (>>==) x f = (fun nr_four -> x >>= f nr_four)
- let (>>==) x f nr_four = (x >>= f nr_four)
- let (>>==) x f a = (x >>= f a)

- great.. now define one for each number of arguments?

Towards nicer notations

- It's not too common to introduce >>==
- People have invented tons of 'nice' notations for >>=
- It helps to realize that there is a pattern:
- Some 3 >>= plus1 >>= double >>= double

- is the same as:

- Some 3 >>= (fun nr3
  -> plus1 nr3 >>= (fun nr4
  -> double nr4 >>= (fun nr8
  -> double nr8)))

Towards nicer notations

- It's not too common to introduce >>==
- People have invented tons of 'nice' notations for >>=
- What its use boils down to is a pattern:
- Some 3 >>= plus1 >>= double >>= double

- is the same as:

- Some 3 >>= (fun nr3
  -> plus1 nr3 >>= (fun nr4
  -> double nr4 >>= (fun nr8
  -> double nr8)))

This doesn't look nice at first,
but it resembles "let" notation!!!
let nr3 = Some 3 in
let nr4 = plus1 nr3 in
let nr8 = double nr4 in
double nr8

Towards nicer notations

- This pattern works just fine for multiple argument functions!

- Some 3 >>= (fun nr3
  -> plus1 nr3 >>= (fun nr4
  -> Some 6 >>= (fun nr6
  -> plus nr4 nr6)))

- (think: let nr3 be the result of Some3, let nr4 be … etc)

Towards nicer notations

- This pattern also works when arguments are used twice!

- Some 3 >>= (fun nr3
  -> plus1 nr3 >>= (fun nr4
  -> plus nr3 nr4 >>= (fun nr7
  -> plus nr4 nr7)))

Crucial observation

- If you look up the definition of >>=
  and try to fit it all into what the program is doing…
- … the result is a very big complex program

- Instead, think of e1 >>= (fun x -> e2) as a special case of 'let x = e1 in e2'
  however, the ('a option) values are bound as 'a:
  This means: e1 is 'a option, but x is 'a
  (which makes this 'let' a bit special)

Towards nicer notations

- ocaml 4.08 allows us to define our own “let*”
- some CSE machines are at 4.08 but I’m not sure if all of them are… Here’s what we could write:

- Some 3 >>= (fun nr3
  -> plus1 <u>nr3</u> >>= (fun nr4
  -> plus <u>nr3</u> <u>nr4</u> >>= (fun nr7
  -> plus <u>nr4</u> nr7)))

- let (let*) = (>>=)
- let* nr3 = Some 3 in
  let* nr4 = plus1 nr3 in
  let* nr7 = plus nr3 nr4 in
  plus nr4 nr7

Back to that generalization…

- Note that >>= can be defined for other types..
- The program below still reminds us of the option monad because we used ‘Some’
- People like to abstract away from that too, defining:
  let return x = Some x

- Some 3 >>= (fun nr3
  -> plus1 nr3 >>= (fun nr4
  -> plus nr3 nr4 >>= (fun nr7
  -> plus nr4 nr7)))

- return 3 >>= (fun nr3
  -> plus1 nr3 >>= (fun nr4
  -> plus nr3 nr4 >>= (fun nr7
  -> plus nr4 nr7)))

What are monads?

- Monads are a variation on pipelining
- The main workhorse of monads is called ‘bind’, written >>=, the other part is ‘return’.
- The option monad (aka maybe monad) defines ‘bind’ and ‘return’ like we did today.

- If you want to say you defined a monad, your functions bind and return should be pipeline-like enough
- Commonly, that means they should intend to satisfy:
- (return r >>= f) = f r
- (v >>= return) = v
- ((v >>= (fun x -> g x)) >>= h) = (v >>= (fun x -> (g x >>= h)))

On Monads …

- If you wish to define bind (>>=), go ahead!
- bind is not defined in the standard library,
  but some commonly used libraries provide them.
- There's one for option, and many more…
- Your TAs have not been taught 'monads'
  - I won't be creating homework that *requires* you to use them: you can do all midterms / assignments without >>=
  - I encourage you to try it out because it might help you (and maybe teach your TAs a thing or two)
  - … and because I want my next TAs to know them :-)