Summary

Legend

- Yellow coloured words or sentences are key words;
- Green coloured words are jargon terms;
- Blue coloured words are copy and paste and is unknown what it means;
- Red coloured words may be wrong *.
- Function composition is saying that a part of the lecture has not be made a summary of, as it isn't yet understood.

* everything may be wrong… XD?

Lesson 1

Theory

The computational model is essentially based on types (primitives (int, float, string)) and functions. The basic on

```
f : a -> b
```

F is the name of the function, a is the input type and the return value.

```
let incr = Fun<number,number>(x => x + 1)
```

The function above should be read as:

```
Let name = Fun<input,output>(body)
```

Primitive types can be combined (or used) to get a new outcome. I.e. 5 + 2 = 10.

Function composition is combing two functions together. I.e. function F and G could be combined. This is often done by creating a new function which feeds its input to F and gives the output of F to G. The output of G becomes the output of the new function that combined the two. Note: that the functions do need to be compatible. Input variable for the function will be ‘a’ and will be given to fun ‘f’ which output ‘b’. Fun ‘g’ will take the outcome of fun ‘f’ which is ‘b’ and will produce ‘c’.

Identity function quote from GrandeOmega:

> The uniqueness of the identity function is its behavior with respect to composition. Composing the identity function with any other function will produce no change whatsoever, meaning that f.then(id()) and id().then(f) are exactly the same as just f.

Referential transparency is to ensure that a function works on every value. I.e. doesn’t matter if a 4 or 5 is given the function will work. If a function has a counter that only runs after the function is called 1000 times, then the function is not referential transparent as conditions depend the outcome of the test.

Questions/observation

- Understand it all expect the Identity functions not completely

Lesson 2

Lesson

This lecture is about structure preserving transformation.

In statically typed programming like TypeScript ensures that nothing else exists outside the given data types, hence data type structed the application.

A container is a strictly defined predetermined structure that cannot be altered. There are containers with a generic structure which the type of a value (i.e. number or string) isn't yet defined.

Transforming containers can transform a value of a container. If a container has a value named A (generic in this example) and another value named B, then chancing the value A into another value will mean creating a new container. Important is to keep the other data, meaning keeping the value named B. I.e. container a with values content: 'A' and counter: 3 will become a container containing: content: False and counter: 3. Often this is called a map container.

Composition of functors is creating something new that is based entirely on the two original functors.

Pipeline is using the 'then' function.

Generalization is type F (or Unit?) use as a placeholder of a function. Somewhat similar then identity.

Distribution is composing two Funs (then) to a map.

other aspects are not (yet) covered within this document.

Code

```
//simple container
type Point = { x: number, y: number}
//generic structure
type Countainer<a> = { content:a, counter:number }
```

```
//e.g. how container could be used
let c_i:Countainer<number> = { content:10, counter:0 }
let c_s:Countainer<string> = { content:"Howdy!", counter:0 }
let c_ss:Countainer<Array<string>> = { content:["Howdy", "!"], counter:0 }
```

```
let map_countainer =
  function<a,b>(f:Fun<a,b>, c:Countainer<a>)
  : Countainer<b> {
  return { content:f.f(c.content), counter:c.counter }
}
```

```
//interface?
type map_countainer = <a,b>(f:Fun<a,b>, c:Countainer<a>) => Countainer<b>
```

Questions/observation

- Don’t understand the code *map_f* and *identity*? Unable to get it to work.

Lesson 3

Lesson

Datatypes support some operations, as example number support the “+” which allows the following result: a + b = c. Also, arrays of ‘arbitrary’ types support this. Generalization to apply operations on datatypes is called: monoid. I.e. a <+> (b <+> c) = (a <+> b) <+> c. Associative Laws means that the order op + or * doesn’t matter.

Monoids

Code

```
type Id<a> = a
let map_Id = function<a,b>(f:Fun<a,b>) : Fun<Id<a>, Id<b>> { return f }
```

Questions/observation

- Functors are simply functions?
- What is the “concat” operation?
- Monoid definition code is unknown how to implement it?

Lesson 4

Lesson

Functors can have a monoidal structure meaning having unit and join functions.

The benefits of monads are:

- exception handling;
- list comprehensions;
- state management;
- concurrency;
- backtracking ("classic AI").

Some operators of monads are bind and Kleisli. When the when the return value is encapsulated in an instance of a monad, is also known as Kleisli composition.

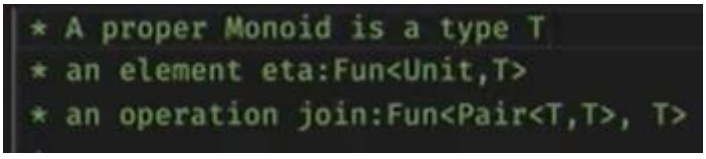
GrandeOmega says about bind:

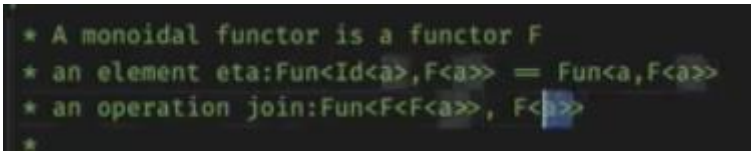
> The attentive reader might have noticed that this pattern is very common, in this very same format, in the JavaScript world: `Promise`, the fundamental data structure used to perform remote calls, has a `then` method which is used exactly as we just described. The only minor difference is that instead of `unit`, we would then use `Promise.resolve`, which has the very same meaning but another name.
>
> The bind operator allows us to merge together (the "content") of a monad to a function which turns that content into another monad. The result of this merging operation is a monad itself.

Pair function of monads are like containers.

Notes from video:

Monad is in domain of simple things. I.e. is type T (i.e. number) – 1+1 =2.





Code

…

Questions/observation

- What is pair function in nomads?

Lesson 5

Lesson

Don’t take a ‘dangerous’ assumption by assuming that code always works.

This lecture contains option, some and none. Combining them with the monadic operators’ unit_Option and join_option.

Grandeomega talking about things like option in ‘mainstream leagues’:

> .Net has had Nullable for quite a while, Option is available in Rust, ML, Maybe is part of Haskell, Java has seen Optional added to the latest versions of the standard library.

Code

…

Questions/observation

```
let inl = <a,b>() : Fun<a, Either<a,b>> => Fun(a => ({ kind:"left", value:a }))
```

Lesson 6 & 7

Lesson

A lot of practical material to learn. This lecture focus on modelling the dynamic part of programs instead of the static part which the focus has been so far during this course. Focusing on what a program can do a not what a program is.

Code

…

Questions/observation

…

Lesson 8

Grandeomega:

> In this chapter, we will define the last extension to our process modeling framework, so that processes become interruptible. Interruptible processes do not necessarily run until completion: they may choose to pause themselves, therefore better integrating with interactive applications. During an interruption of such a process, the application will be able to update animations, handle user input, and so on.

Coroutines are known for only having three possible outcomes: a result, an error, or an interruption.

Lesson

…

Code

…

Questions/observation

…

Typescript

Create new NPM project:

```
Run: Npm init
Run: Npm install typescript
Make a typescript file
Run: .\node_modules\.bin\tsc .\main.ts -w
Run another NPM: node .\main.js
```