

FIT2102

Programming Paradigms

Lecture 4

Closures and Continuation Programming

Lazy Evaluation

Functional Reactive Programming

Faculty of Information Technology



MONASH
University

Safety First

Abstracting common error-prone code patterns into robust, composable functions

Type safety



Learning Outcomes

- Compare the goals and abstractions of Object Oriented Programming with Functional Programming
- Create ES6 classes and describe their relationship with prototype based classes in ES5
- Explain how ES6 classes can be used to provide a fluent interface and compare this to function chaining
- Create programs using closures to achieve:
 - Continuations
 - Callback event handling
- Explain the Observable pattern and create asynchronous programs using Functional Reactive Programming

Object Oriented versus Functional Programming

- OO is a programming paradigm that seeks to model data entities with *class* types, *objects* are concrete instances of a class.
- *Encapsulation* is the principle of limiting access to a class's members to achieve *separation of concerns* and limit the ways that object's state can be mutated.
- Functional Programming seeks to limit complexity by separating data from functions that manipulate the data.
- In FP we restrict mutation of data and other side effects through function purity.
- These goals are not mutually exclusive and we can successfully combine FP and OO principles.

	Functional	Object-Oriented
Unit of Composition		
Programming Style		
Data and Behaviour		
State Management		
Control Flow		
Thread Safety		
Encapsulation		

The above is a traditional view from the perspective of FP vs OO languages.

Modern languages like TypeScript, Scala, C++17, Java 8 etc, provide facilities for both paradigms for you to use (or abuse) as you please.

	Functional	Object-Oriented
Unit of Composition	Functions	Objects (classes)
Programming Style	Declarative	Imperative
Data and Behaviour	Loosely coupled through pure, generic functions	Tightly coupled in classes with methods
State Management	Treats objects as immutable	Favours mutation of objects through instance methods
Control Flow	Functions, recursion and chaining	Loops and conditionals
Thread Safety	Pure functions easily used concurrently	Can be difficult to manage
Encapsulation	Less essential	Needed to protect data integrity

The above is a traditional view from the perspective of FP vs OO languages.

Modern languages like TypeScript, Scala, C++17, Java 8 etc, provide facilities for both paradigms for you to use (or abuse) as you please.

Creating Objects from a Prototype in ES5 (JavaScript 1.5)

```
function Person(name, occupation) {  
    this.name = name  
    this.occupation = occupation  
}  
Person.prototype.sayHello = function() {  
    console.log(`Hi, my name's ${this.name} and I'm ${this.occupation}!`)  
}  
const tim = new Person("Tim", "a skater dude")  
tim.sayHello()  
> Hi, my name's Tim and I'm a skater dude!
```

Note: with arrow syntax `this` refers to the enclosing scope, rather than the function object, i.e. can't use `=>` to declare methods that operate on the object

Creating Objects from a Prototype in ES5 (JavaScript 1.5)

The `prototype` is a special property on functions that is used by objects instantiated from that function by a `new` invocation of the function's constructor

```
function Person(name, occupation) {  
  ...  
}  
const tim = new Person("Tim", "a skater dude")
```

If the JavaScript engine cannot find a property on such an object it looks for it on the object's prototype - thus, such properties are available to all instances.

It's effectively a class mechanism with *static* and *instance* members.

TypeScript (and ES6) make all this more clear.

Week 3 Tute Exercise 4 - wrapping cons list

TypeScript Class Definition:

```
class List<T> {  
    private head: Cons<T>;  
  
    constructor(list: T[] | Cons<T>) {  
        this.head = list instanceof Array ?  
            fromArray(list) : list;  
    }  
  
    map<V>(f: (item: T) => V): List<V> {  
        return new List(map(f, this.head));  
    }  
  
    ...  
}
```

Generated JavaScript (ES5):

```
var List = (function () {  
  
    function List(list) {  
        this.head = list instanceof Array ?  
            fromArray(list) : list;  
    }  
  
    List.prototype.map = function (f) {  
        return new List(map(f, this.head));  
    };  
  
    ...  
    return List;  
}());
```

⇨ Immediately Invoked Function Expression (IIFE)

Static vs Non-static Member functions

TypeScript:

```
class List<T> {  
    ...  
    static concat<T>(u: ConsList<T>, v: ConsList<T>)  
        : ConsList<T> {  
        return u ? cons(head(u), List.concat(rest(u), v))  
            : v ? List.concat(v, undefined)  
            : undefined  
    }  
    concat(b: List<T>): List<T> {  
        return new List(  
            List.concat(this.head, b.head));  
    }  
    ...  
}
```

Generated JavaScript (ES5):

```
var List = (function () {  
    ...  
    List.concat = function (u, v) {  
        return u ? cons(head(u), List.concat(rest(u), v))  
            : v ? List.concat(v, undefined)  
            : undefined;  
    };  
  
    List.prototype.concat = function (b) {  
        return new List(  
            List.concat(this.head, b.head));  
    };  
    ...  
    return List;  
})();
```

Chaining and fluent interfaces

Function chaining:

```
reduce((t, s)=>(!t[s] ? t[s] = 1 :  
    t[s]++, t), <any>{},  
    map(s => s.toUpperCase(),  
        filter(s => s[0] != 't',  
            fromArray([  
                'Tim', 'sally',  
                'Anne', 'sally'  
            ]))))
```

Fluent Interface:

```
const tally = new List([  
    'tim', 'sally', 'anne', 'sally'])  
    .filter(s => s[0] != 't')  
    .map(s => s.toUpperCase())  
    .reduce((t, s)=>  
        (!t[s] ? t[s] = 1 : t[s]++, t),  
        ({}))
```

Eager versus Lazy Evaluation

```
const eagerDate = Date();  
const lazyDate = function() {  
  return Date();  
}  
  
function printTime() {  
  console.log("Eager: " + eagerDate)  
  console.log("Lazy: " + lazyDate())  
}  
  
setTimeout(printTime, 1000)
```

TRIVIAL

But interesting!

```
> Eager: Sat Aug 12 2017 20:27:13 GMT+1000 (AUS Eastern Standard Time)
```

```
> Lazy: Sat Aug 12 2017 20:27:14 GMT+1000 (AUS Eastern Standard Time)
```

Eager evaluation

JavaScript (and all imperative languages) evaluates expressions eagerly.

Is this useful?

```
function sillyNaturalNumbers(initialValue:number): number {  
    return sillyNaturalNumbers(initialValue + 1);  
}
```

Lazy Evaluation

By wrapping an expression in a function, we delay its execution until we invoke the returned function:

```
function slightlyLessSillyNaturalNumbers(v) {  
    return () => slightlyLessSillyNaturalNumbers(v+1)  
}
```

We've solved the infinite recursion!


But this is only slightly less silly because we have no way of getting the numbers out.

Lecture Activity

To be announced...

Laziness with Types

```
interface LazySequence<T> {  
    value: T;  
    next(): LazySequence<T>;  
}
```

```
function naturalNumbers(): LazySequence<number> {  
    return function _next(v:number) {  
        return {  
            value: v,  
            next: () => _next(v+1)  
        }  
    } (1)  IIFE  
}
```

```
const n = naturalNumbers()  
n.value  
> 1  
n.next().value  
> 2  
n.next().next().value  
> 3
```


Imperative Event Handling

```
function mousePosEvents () {  
    const pos = document.getElementById("pos");  
    document.addEventListener("mousemove", e => {  
        const p = e.clientX + ', ' + e.clientY;  
        pos.innerHTML = p;  
        if (e.clientX > 400) {  
            pos.classList.add('highlight');  
        } else {  
            pos.classList.remove('highlight');  
        }  
    });  
}
```

Functional Reactive Programming (FRP)

Observable is an interface gradually making its way into the ECMAScript standard library:

“The Observable type can be used to model push-based data sources such as DOM events, timer intervals, and sockets. In addition, observables are:

- Compositional: Observables can be composed with higher-order combinators.
- Lazy: Observables do not start emitting data until an observer has subscribed.”

Functional Reactive Programming (FRP)

```
function mousePosObservable() {  
  const  
    pos = document.getElementById("pos"),  
    o = Observable  
      .fromEvent<MouseEvent>(document, "mousemove")  
      .map(({clientX, clientY})=> ({x: clientX, y: clientY}));  
  
  o.map(({x,y}) => `${x},${y}`)  
    .subscribe(s => pos.innerHTML = s);  
  
  o.filter(({x}) => x > 400)  
    .subscribe(_ => pos.classList.add('highlight'));  
  
  o.filter(({x}) => x <= 400)  
    .subscribe(_ => pos.classList.remove('highlight'));  
}
```

constructs a stream with
three branches:

Observable<x,y>

|- set innerHTML

|- add highlight

|- remove highlight

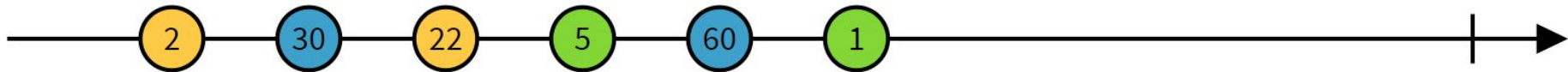
Observable.map



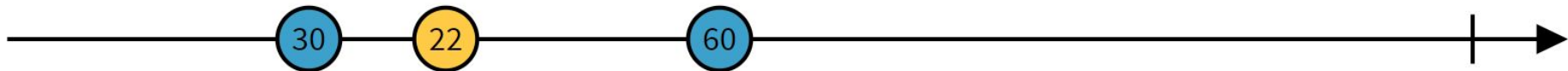
```
map(x => 10 * x)
```



Observable.filter



```
filter(x => x > 10)
```



Cleaning up events

```
/**
 * animates an SVG rectangle, passing a continuation to the built-in HTML5 setInterval function.
 * a rectangle smoothly moves to the right for 1 second.
 */
function animatedRectTimer() {
  const svg = document.getElementById("animatedRect");
  let rect = new Elem(svg, 'rect')
    .attr('x', 100).attr('y', 70)
    .attr('width', 120).attr('height', 80)
    .attr('fill', '#95B3D7');
  const animate = setInterval(()=>rect.attr('x', 1+Number(rect.attr('x'))), 10);
  const timer = setInterval(()=>{
    clearInterval(animate);
    clearInterval(timer);
  }, 1000);
}
```

Cleaning up with Observable

```
/**
 * Demonstrates the interval method on Observable.
 * The observable stream fires every 10 milliseconds.
 * It terminates after 1 second (1000 milliseconds)
 */
function animatedRect() {
  const svg = document.getElementById("animatedRect");
  let rect = new Elem(svg, 'rect')
    .attr('x', 100).attr('y', 70)
    .attr('width', 120).attr('height', 80)
    .attr('fill', '#95B3D7');

  Observable.interval(10)
    .takeUntil(Observable.interval(1000))
    .subscribe(()=>rect.attr('x', 1+Number(rect.attr('x'))));
}
```

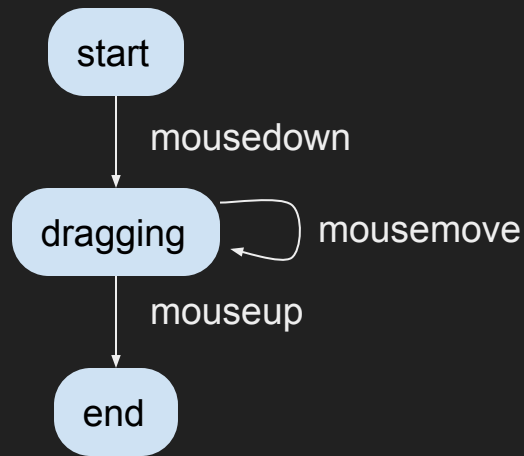
Observable.takeUntil



takeUntil




```
function dragRectEvents() {  
  const svg = document.getElementById("dragRect");  
  const rect = new Elem(svg, 'rect')  
    .attr('x', 100).attr('y', 70)  
    .attr('width', 120).attr('height', 80);  
  rect.elem.addEventListener('mousedown', (e:MouseEvent)=> {  
    const  
      xOffset = Number(rect.attr('x')) - e.clientX,  
      yOffset = Number(rect.attr('y')) - e.clientY,  
      moveListener = (e:MouseEvent)=>{  
        rect  
          .attr('x',e.clientX + xOffset)  
          .attr('y',e.clientY + yOffset);  
      },  
      done = ()=>{ svg.removeEventListener('mousemove',moveListener); };  
    svg.addEventListener('mousemove', moveListener);  
    svg.addEventListener('mouseup', done);  
    svg.addEventListener('mouseout', done);  
  })  
}
```



Observable.flatMap

```
Observable.fromArray([1,2,3])  
  .flatMap(x=>Observable.fromArray([4,5,6])  
    .map(y=>[x,y]))  
  .subscribe(console.log)
```

[1, 4]

[1, 5]

[1, 6]

[2, 4]

[2, 5]

[2, 6]

[3, 4]

[3, 5]

[3, 6]

[1,2,3]

| fromArray

O<number>

| flatMap

1-----2----- ...

O<number>

| map

[[1,4],[1,5],[1,6]]

|

+-----+----- ...

O<number>

| map

[[2,4],[2,5],[2,6]]

|

O<number[]>

| subscribe(console.log)

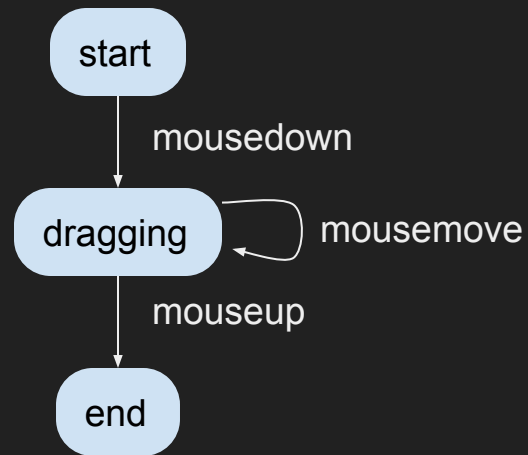
[1,4]

[1,5]

...

```

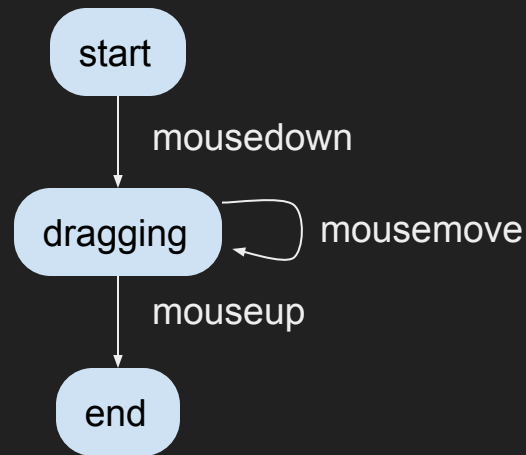
0<MouseDown>
  | map x/y offsets
0<x,y>
  | flatMap
+-----+----- ...
0<MouseMove>          0<MouseMove>
  | takeUntil mouseup  |
0<MouseMove>          0<MouseMove>
  | map x/y + offsets  |
+-----+----- ...
0<x,y>
  | move the rect
---
```



```

function dragRectObservable() {
  const svg = document.getElementById("dragRect"),
    mousemove = Observable.fromEvent<MouseEvent>(svg, 'mousemove'),
    mouseup = Observable.fromEvent<MouseEvent>(svg, 'mouseup');
  const rect = new Elem(svg, 'rect')
    .attr('x', 100).attr('y', 70)
    .attr('width', 120).attr('height', 80);
  rect.observe<MouseEvent>('mousedown')
    .map(({clientX, clientY}) => ({
      xOffset: Number(rect.attr('x')) - clientX,
      yOffset: Number(rect.attr('y')) - clientY
    })))
    .flatMap(({xOffset, yOffset}) =>
      mousemove
        .takeUntil(mouseup)
        .map(({clientX, clientY}) => ({
          x: clientX + xOffset,
          y: clientY + yOffset
        })))
    .subscribe(({x, y}) => rect.attr('x', x).attr('y', y));
}

```



Conclusions

FRP brings together many of the concepts we have been discussing:

- List data flow processing
- Lazy evaluation
- Fluent Interfaces
- Compare to another application of anonymous functions: event handlers
- Allows sequential and asynchronous stream operations to be implemented in linear code