Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Τμήμα Πληροφορικής & Τηλεπικοινωνιών

Προηγμένες Μέθοδοι Προγραμματισμού

ΠΜΣ (M135.CS1E, M135.CS23B, M135.IC1E, παλαιό: M117)

Functional Programming

Δρ. Κώστας Σαΐδης (saiko@di.uoa.gr)

Functional programming

The function is the dominant form of abstraction.

Key concepts

- Higher-order functions
- Lambdas (anonymous functions)
- Pure functions (referential transparency): functions invoked with the same arguments return the same results (no side-effects)
- Closures
- Immutability (no state mutations)
- (Tail-)Recursion (instead of iteration)
- Function composition, partial functions & currying

Function

A language construct that

- names a block of code, which
  - accepts some arguments
  - returns a value

Function declaration

```
function sum(a, b) {
  return a + b;
}
```

All examples are in Javascript.

Higher-order function

Some languages treat a function as an ordinary value:

- it can be assigned to a variable,
- it can be passed as argument to another function,
- it can be returned by another function.

E.g. a callback, a promise then / catch / finally handler, etc.

Lambda function

An anonymous function, usually treated as a value (assigned to variables, passed to other functions, etc)

Function expression (lambda)

```
const sum = function(a, b) {
  return a + b;
};
```

Arrow function (lambda)

```
const sum = (a, b) => a + b;
```

IIFE

```
((a, b) => a + b)(3, 4);
```

Pure function

- A function that, given an input, will always return the same output.
- It does not depend on any context / environment / state.
- It has no side-effects (this is the actual equivalent to a mathematical function).

```
x => x + x
(a, b) => a + b
```

A pure function is deterministic. It can be easily parallelized. It can be formally proved to be correct.

Non-pure functions

```
function(url, callback) => {
  ajax.get(url).then(callback);
}
```

```


let x = 2;

function add(y) {
  return y + x;
}
```

Closure

A function along with its outer lexical environment (the scope in which it was defined).

- Free variables are allowed in the function body.
- They are bound to their outer lexical environment.
- The closure holds a "live" reference to its scope/environment.

Example

```javascript
let name = "John";

function sayHi() {
  alert("Hi, " + name);
}

name = "Pete";

sayHi();
```

From https://javascript.info (have a look, it's pretty neat).

Another example

```
function makeCounter() {
  let cnt = 0;

  return function() {
        return cnt++;
  }
}

const inc = makeCounter();
inc(); //1
inc(); //2
```

The emphasis on functions leads to the following shift (compared to imperative programming):

- argument passing (instead of variable assigning)
- (tail-)recursion (instead of iteration)
- immutability
- no looping

Examples

List operations (filter, map, fmap, reduce, etc)

Javascript Arrays

Groovy Collections

Swiss army knife (reduce)

You can implement any list transformation with reduce.

```
list.reduce(accumulator function, initial value of accumulator)
accumulator function = (current value of accumulator, current element) => new value of accumulator
```

Currying

Transform the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

```
const add = (x,y) => x + y;
const add2 = x => y => x + y;
```

```
function add2(x) {
  return function(y) { // closure here
    return x + y;
  }
}
```

Invoke as `add2(3)(4)`

Partial functions

Currying allows us to pass the arguments at different points in time (something like a function builder).

```
// a helper function
const authorize = (user, action) => action | null;

// an authentication filter
const user = new User(username);
const actionAuthorizer = authorize.curry(user);
// a partially applied function, returns a function with a single arg (the action)

// an authorization filter
const action = new Action("deleteEverything")
actionAuthorizer.apply(action);
```

Function currying in JS

Functors

A type that can be mapped over (has some sort of a map function, a "Mappable")
according to the following laws:

1. Identity

```
functor.map(x => x) === functor
```

2. Composition is chaining

```
functor.map(x => f(g(x))) === functor.map(g).map(f)
```

What?

Functors abstract the container away and allow chaining.

Javascript arrays are functors.

Promises are functors.

Java streams are functors.

We can define a functor out of (almost) any value.

Example

```
class SingleValueFunctor {
  constructor (value) {
    this.value = value
  }
  map (f) {
    return new SingleValueFunctor(f(this.value))
  }
}
```

And much more

- Applicatives
- Monoids
- Monads

Not covered by this course.

A nice place to start reading more