Advanced Programming

Final Term Paper

Andrea Madotto

Matricola:540794

Computer Science Department
University of Pisa

Exercize 1

The object model of a component is an object with the following fields and methods: a Constructor method, that as a default value takes an empty function, a render method which is also a function and it has as default value a function that returns an empty sting, all the fields provided as input, and finally all the generated fields by the invocation of the Constructor methods defined before. In the following an implementation of React.class and React.render.

```
function Component(obj) {
    this.constructor = function() {};
    this.render = function() {
        return '';
    };
    for (var key in obj) {
        this[key] = obj[key];
    }
    this.old = null;
}

var React = (function() {
    var generateHTML = function(node) {
        if (typeof node !== "object") {
            return node;
        } else if (node instanceof Array) {
            var elem = "";
            for (var i = 0; i < node.length; i++) {
                elem += generateHTML(node[i]);
            }
            return elem;
        } else {
            var e = '';
            e += "<" + node.tag + " ";
            for (var key in node.attrs) {
                e += key + "='" + node.attrs[key] + "'";
            }
            e += ">";
            e += generateHTML(node.children);
            e += "</" + node.tag + ">";
            return e;
        }
    };
    return {
        class: function(obj) {
            var comp = new Component(obj);
            comp.constructor();
            return comp;
        },
        render: function renderer(component, DOM) {
            var virtualdom = component.render();
            if (!(virtualdom instanceof Component)) {
                DOM.innerHTML = generateHTML(virtualdom);
            } else {
                renderer(virtualdom, DOM);
            }
        }
    };
})();
```

Listing 1: Simplest definition of react component and a basic render method

Notice that: in the Component we already include an attribute ”old” that is going to useful in the next exercises, and in this basic render, we didn't include the connection between function and methods. Indeed, we have just print out the content of the corresponding function, just to write something inside the attribute.

Exercise 2

In order to optimize the rendering function we will use the field "old", belonging to the component, to keep track of the previous rendered VDOM. The latter is a lightweight structure that has the same shape of a DOM, a tree, but it's maintained by JS code. Therefore we are going to update the render method, of the class React as follow.

```
render: function renderer(component, DOM) {
    var virtualdom = component.render();
    if (!(virtualdom instanceof Component)) {
        if (component.old === null) {
            DOM.innerHTML = generateHTML(virtualdom);
        } else {
            if (diff(component.old, virtualdom) !== undefined) {
                patch(virtualdom, DOM.children);
            }
        }
        component.old = JSON.parse(JSON.stringify(virtualdom));
    } else {
        renderer(virtualdom, DOM);
    }
}
```

Listing 2: New render method

The first time we render a component, we simply call *generateHTML* in all the VDOM. The second time, instead of re-render all the VDOM, we make use of the old VDOM (saved in the component and update at each render) to modify just the part of the DOM that has been really changed. Therefore, we employ two functions: *diff* and *patch*. The first go trough the old VDOM and the current VDOM (with the updated values), to mark whether a node should be re-render or not. Furthermore, *diff* is going to flag the node with different marker, using an "enum" (conceptually), based on the kind of operation that should be done to update the DOM. In the following we have the code:

```
var fl = {
    LEAF    : 1, NEWNODE: 2, MARKER : 3,
    TAG     : 4, ATT     : 5, DEL     : 6
};
var diff = function(oldVdom, newVdom) {
    switch (type(oldVdom, newVdom)) {
        case "array":
            for (var i = 0; i < newVdom.length; i++) {
                var val = diff(oldVdom[i], newVdom[i]);
                if (val) {
                    newVdom.flag = fl.MARKER;
                    return val;
                }
            }
            break;
        case "object":
            if (oldVdom.tag !== newVdom.tag) {
                newVdom.flag = fl.TAG;
                return fl.MARKER;
            } else {
                for (var key in newVdom.attrs) {
                    if (key !== "onClick" && newVdom.attrs[key] !== oldVdom.
                        attrs[key]) {
                        newVdom.flag = fl.ATT;
                        return fl.MARKER;
                    }
                }

            }
            var childVal = diff(oldVdom.children, newVdom.children);
            if (childVal === fl.LEAF || childVal === fl.NEWNODE ||
                childVal === fl.MARKER || childVal === fl.DEL) {
                newVdom.flag = childVal;
                return fl.MARKER;
            }
            break;
        case "literal":
            if (oldVdom !== newVdom)
                return fl.LEAF;
            break;
        case "del":
            return fl.DEL;
        case "ins":
            return fl.NEWNODE;
    }
};
var type = function(oldVdom, newVdom) {
    if (oldVdom instanceof Array && newVdom instanceof Array) {
        if (oldVdom.length > newVdom.length)
            return "del";
        else if (oldVdom.length < newVdom.length)
            return "ins";
        else
            return "array";
    } else if (typeof oldVdom === "object" && typeof newVdom === "object")
        return "object";
    else if (typeof oldVdom === "string" && typeof newVdom === "string" ||
        typeof oldVdom === "number" && typeof newVdom === "number")
        return "literal";
    else
        return null;
};
```

Listing 3: *diff* function

After the *diff* function, that works in the VDOM, we will know whether we need to modify the real DOM or not, and exactly where. Indeed if the function return undefined, it means that nothing has been change and we don't need to patch the DOM. Otherwise, the return of the *diff* is a number, and then we call *patch*. The *patch* function is going through the labelled VDOM and the real DOM. Notice that it will not go through all the tree, but just in the part that has been labelled. Here the code of the function.

```
var patch = function(newVdom, DOM) {
    if (typeof newVdom !== "object") {
        return newVdom;
    } else if (newVdom instanceof Array) {
        for (var i = 0; i < newVdom.length; i++) {
            patch(newVdom[i], DOM[i]);
        }
    } else {
        switch (newVdom.flag) {
            case fl.LEAF:
                DOM.innerHTML = generateHTML(newVdom.children);
                break;
            case fl.TAG:
                DOM.outerHTML = generateHTML(newVdom);
                break;
            case fl.ATT:
                for (var key in newVdom.attrs) {
                    DOM.setAttribute(key, newVdom.attrs[key]);
                }
                break;
            case fl.DEL:
                DOM.outerHTML = generateHTML(newVdom);
                break;
            case fl.NEWNODE:
                DOM.outerHTML = generateHTML(newVdom);
                //var elemtoadd = newVdom.children[newVdom.children.length - 1];
                //var node = document.createElement(elemtoadd.tag);
                //node.innerHTML = generateHTML(elemtoadd.children);
                //DOM.appendChild(node); //explanation is below
                break;
            case fl.MARKER:
                if (typeof newVdom.children === "object" &&
                    !(newVdom.children instanceof Array)) {
                    patch([newVdom.children], DOM.children);
                } else {
                    patch(newVdom.children, DOM.children);
                }
                break;
        }}
};
```

Listing 4: *patch* function

Notice that, we added some special cases in the MARKER label, because the DOM treat the child nodes always as an Array, instead our VDOM (node) it could have a single node or an array of nodes.
We would also like to analyse a limitation that we encountered: we couldn't handle all the possible cases in the optimized version. For example the insertion of a new node in a selected position (e.g. insert a node as i_{th} child). This because, we weren't able to add custom *id* to every node, without modify the given structure (Node). Indeed, the addiction of such *id* in each node require to maintain an additional data structure, that we believe was a bit overkilling in this case. Therefore, each time we labelled the operation as DEL or NEWNODE, we simply render the node and its sub-tree. However, in the code comments we keep a version of insertion that add a new node, as last child, without re-render all the siblings.

Exercise 3

To connect DOM events to the functions in the Component, we will modify two parts of the previous code: Class and Generate HTML. In the first, we add an *id* for each function, to be used as uniquely identifier of that components' functions. In the second, we connected the attribute value with the function. In practise, to handle DOM event, we add the corresponding function to *Window* (named as the identifier previously defined) and we use bind() method to bind the function to the right component. Here the code:

```
class: function(obj) {
    var comp = new Component(obj);
    for (var funName in comp) {
        if (funName !== "constructor" &&
            funName !== "render" &&
            funName !== "old") {
            var funId = 'fun' + index++;
            comp[funName] = {
                name: funId,
                fun: comp[funName]
            };
        }
    }
    comp.constructor();
    return comp;
}
```

Listing 5: New class

```
var generateHTML = function(node, funTemplate) {
    if (typeof node !== "object") {
        return node;
    } else if (node instanceof Array) {
        var elem = "";
        for (var i = 0; i < node.length; i++) {
            elem += generateHTML(node[i], funTemplate);
        }
        return elem;
    } else {
        var e = '';
        e += "<" + node.tag;
        for (var key in node.attrs) {
            var funobj = node.attrs[key];
            (function(funobj) {
                window[funobj.name] = function(e) {
                    funTemplate(funobj.fun, e);
                };
            })(funobj);
            if (!funobj.name) {
                e += " " + key + "='" + node.attrs[key] + "'";
            } else {
                e += " " + key + "='" + funobj.name + "(event)'";
            }
        }
        e += ">";
        e += generateHTML(node.children, funTemplate);
        e += "</" + node.tag + ">";
        return e;
    }
};
```

Listing 6: New generateHTML

Therefore, each time we call generateHTML we will provide the VDOM, and a function f. The latter will be used when the attributes are generated, and must be defined in a scope where component and DOM are defined. Therefore we are going to have:

```
var f = function(f, e) {
    f.bind(component)(e);
    renderer(component, DOM);
};
generateHTML(virtualdom, f);
```

Listing 7: how to call generate HTML

To see the ”counter” example in action, click the following link to JSfiddle page: JSFIDDLE

Exercise 4

First of all we define a Tokenizer class, with two accessible methods: *next* and *get*.

```
var TOKEN = {
  ANGOPEN   : 1, ANGCLOSE   : 2, GRAPHOPEN: 3, GRAPHCLOSE: 4,
  TAGCLOSE1: 5, TAGCLOSE2 : 6, EQUAL     : 7, QUOTE      : 8
};

function Stream(string) {
  var p = 0;
  this.peak = function () {
    return string.charAt(p);
  };
  this.consume = function () {
    ++p;
  };
}

function Tokenaizer(stream) {
  var current_token = '';
  var remove_space = function () {
    while (stream.peak() && stream.peak() == ' ')
      stream.consume();
  };

  var parse_string = function () {
    remove_space();
    var str = '';
    while (stream.peak() &&
      stream.peak() != ' ' &&
      stream.peak() != '>' &&
      stream.peak() != '<' &&
      stream.peak() != '/' &&
      stream.peak() != '}' &&
      stream.peak() != '{' &&
      stream.peak() != '=' &&
      stream.peak() != '"') {
      str += stream.peak();
      stream.consume();
    }
    return str;
  };

  this.next = function () {
    remove_space();
    switch (stream.peak()) {
      case '<':
        stream.consume();
        if (stream.peak() == '/') {
          stream.consume();
          return current_token = TOKEN.TAGCLOSE1;
        } else
          return current_token = TOKEN.ANGOPEN;
      case '/':
        stream.consume();
        if (stream.peak() == '>') {
          stream.consume();
          return current_token = TOKEN.TAGCLOSE2;
        } else
          throw new Error("Wrong character after '/'");
      case '>':
        stream.consume();
        return current_token = TOKEN.ANGCLOSE;
      case '{':
        stream.consume();
        return current_token = TOKEN.GRAPHOPEN;
      case '}':
        stream.consume();
        return current_token = TOKEN.GRAPHCLOSE;
      case '=':
          stream.consume();
          return current_token = TOKEN.EQUAL;
      case '"':
        stream.consume();
        return current_token = TOKEN.QUOTE;
      case undefined:
        return undefined;
      default:
        return current_token = parse_string();
    }
  };

  this.get = function () {
    return current_token;
  };
}
```

Listing 8: Tokenizer

Then, we are going to create the recursive descending parser. In doing so, we manipulate the grammar in order to remove ambiguity.

```
jsxel    := <tag jsxelrec | jsxval
jsxattr  := [name=jsxval]*
jsxrec   := jsxattr> jsxel* tagClose | />
tagClose := </tag>
tag      := string
name     := string
jsxval   := " string " | { JS }
JS       := JavaScript expression
```

Therefore, we are going to create a function for every term of the grammar to be match. The output of the parser is going to have the same structure of the node described in the text. Indeed, we created also a class Node.

```
var Node = function(tag) {
    this.tag = tag;
};

var parser = function() {

    this.parse = function(text) {
        var nodes = [];
        var stream = new Stream(text);
        T = new Tokenaizer(stream);
        T.next();
        while (T.get()) {
            nodes.push(jsxel());
        }
        return nodes;
    };

    function jsxel(n) {
        if (T.get() === TOKEN.ANGOPEN) {
            T.next();
            if (typeof T.get() === "string") {
                var node = new Node(T.get());
                T.next();
                return jsxrec(node);
            }
        } else {
            return jsxval();
        }
    }

    function jsxrec(node) {
        if (T.get() === TOKEN.TAGCLOSE2) {
            T.next();
            return node;
        } else {
            jsxattr(node);
            if (T.get() === TOKEN.ANGCLOSE) {
                T.next();
                if (T.get() === TOKEN.TAGCLOSE1) {
                    tagClose(node);
                } else {
                    while (T.get() !== TOKEN.TAGCLOSE1) {
                        var val = jsxel(node);
                        if (val.type) {
                            node.children = val.value;
                        } else {
                            if (!node.children)
                                node.children = [];
                            node.children.push(val);
                        }
                    }
                    tagClose(node);
                }

            } else {
                throw new Error("missing closing tag");
            }
            return node;
        }
    }

    function jsxattr(node) {
        var attr = {},
            attrname, attrvalue;
        while (T.get() !== TOKEN.ANGCLOSE) {
            if (typeof T.get() === "string") {
                attrname = T.get();
                T.next();
                if (T.get() === TOKEN.EQUAL) {
                    T.next();
                    attrvalue = jsxval().value;
                } else {
                    throw new Error('miss equal');
                }
            } else {
                throw new Error('attributt error');
            }
            attr[attrname] = attrvalue;
        }
        if (Object.keys(attr).length !== 0) {
            node.attr = attr;
        }
    }

    function tagClose(node) {
        if (T.get() === TOKEN.TAGCLOSE1) {
            T.next();
            if (typeof T.get() === "string" && T.get() === node.tag) {
                T.next();
                if (T.get() === TOKEN.ANGCLOSE) {
                    T.next();
                } else {
                    throw new Error('missing and >');
                }
            } else {
                throw new Error('tag name is wrong');
            }
        } else {
            throw new Error('missing and </');
        }
    }

    function jsxval() {
        var value;
        if (T.get() === TOKEN.QUOTE) {

            T.next();

            if (typeof T.get() === "string") {
                value = T.get();
                T.next();
                if (T.get() === TOKEN.QUOTE) {
                    T.next();
                    return {type: "string", value: value};
                } else {
                    throw new Error('jsxval string miss');
                }
            } else if (T.get() === TOKEN.QUOTE) {
                T.next();
                return {type: "string", value: ""};
            } else {
                throw new Error('ERROR');
            }
        } else if (T.get() === TOKEN.GRAPHOPEN) {
            T.next();// here we recognise that is a JS expression
            if (typeof T.get() === "string") {
                value = T.get();
                T.next();
                if (T.get() === TOKEN.GRAPHCLOSE) {
                    T.next();
                    return { type: "JS",value: value};
                } else {
                    throw new Error('jsxval string miss');
                }
            } else if (T.get() === TOKEN.GRAPHCLOSE) {
                T.next();
                return { type: "JS", value: ""};
            } else {
                throw new Error('"ERROR"');
            }
        } else {
            throw new Error('wrong jsxval');
        }
    }
};
```

Listing 9: Parser

Exercise 5

Event driven programming is programming paradigm. It controls the flow of program through event such as mouse click, key press etc. Therefore, any time an event will occur, the application will handle the event with an appropriate procedure. The key aspect of an event driven application is a scheduler(event loop) that receives a stream of events and passes each event to the relevant function.

The main benefit of asynchronous programming is the possibility to do not stop the main flow of a program when an synchronous task occur. Indeed, an asynchronous program, when it finds a task that would normally be block by a synchronous program, it will execute some other tasks that can still make progress. So it only blocks task when it can not make any progress. Furthermore, the asynchronous model perform best, compare to a synchronous one, when: the tasks perform a lot of I/O operations, tasks are independent each other, and there are many task (because it is more likely that there is a task to be executed). For example, a server side application that need to handle many connections request, if it uses asynchronous I/O is going to be very scalable.

Nodejs for JavaScript, Twisted for Python, and EventMachine for Ruby are some libraries and frameworks that support event driven programming. Finally, the following two short code represents the reading of a file, one based on asynchronous IO using NodeJS and the other based on multithreading using JAVA8.

```java
ExecutorService executor = Executors.newSingleThreadExecutor();
Future < String > future = executor.submit(() - > {
    try {
        return new String(Files.readAllBytes(Paths.get("SOMEFILE.txt")));
    } catch (IOException e) {
        throw new IllegalStateException("task interrupted", e);
    }
});
try {
    System.out.print("result: " + future.get());
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
```

Listing 10: multithreading file reader using JAVA8

```javascript
var fs = require('fs');
fs.readFile('SOMEFILE.txt', function(err, data) {
   if (err) throw err;
   console.log("result: "+data);
});
```

Listing 11: asynchronous IO file reader using NodeJS