Analysing Functional Paradigm Concepts
The JavaScript Case

Lukáš Janeček(✉) and Robert Pergl

Faculty of Information Technology, Czech Technical University in Prague,
Prague, Czech Republic
{janeclu1,perglr}@fit.cvut.cz

Abstract. Hundreds of programming languages are available today and new ones are still emerging. Nevertheless, they are founded in several (old) paradigms. Knowing the essence of paradigms helps to orient oneself in this Babylon, which is challenging especially for the growing community of programmers with no computer science background. In this paper we focus on functional paradigm, which has a raising attention both in new languages (like Clojure and ClojureScript) and a growing support in traditional languages (like C++ and Java). We do not discuss why this happens here, but we focus on analysing fundamental concepts in the functional paradigm and functional programming languages. We describe them and divide them into two categories: key principles and additional principles. Next, we apply this conceptual framework to analyse the ES5 and ES6 versions of JavaScript. We conclude that ES6 is a good step towards functional principles support. Also, the presented conceptual framework may be used for similar analyses of other languages.

Keywords: Functional programming · Lambda calculus · JavaScript · ECMAScript 5 · ECMAScript 6

1 Introduction and Motivation

There is an astounding number of programming languages available. For example, in the Wikipedia, there is a list more than 700 programming languages [1]. New ones emerge almost every year [1]. However, they are based on a few programming paradigms, being the imperative (or procedural) paradigm, the object-oriented paradigm, logic-based (or rule-based) and the functional paradigm (or applicative) [2]. Programming languages are based on one or more of the paradigms, which they embrace in their own style. There is a growing number of non-professional programmers, e.g. scientists from various fields, who lack the formal computer science education. This Babylon of languages becomes very confusing for them. In this paper we focus on the functional paradigm, which is very old, at the same, it gains popularity nowadays: new languages emerge (like Clojure [3]) and other languages adopt its concepts (C++, Java) [4], while the

Á. Rocha et al. (eds.), *Recent Advances in Information Systems and Technologies*,
Advances in Intelligent Systems and Computing 569, DOI 10.1007/978-3-319-56535-4_86

functional paradigm is strongly rooted in most of the today's mainstream languages, as well (Python, C#, F#, Ruby, Smalltalk and others). This situation is in a strong contrast to awareness of the programmers, where most of them remain with traditional Fortran-style programming (iteration, if-then-else). Our goal is to present the key concepts in the functional paradigm to help programmers see the essence in programming languages.

2 Methodology

We identify key principles from the literature and explain them. We denote the principles by identifiers for easy referencing. We demonstrate their implementation in the popular JavaScript language – the ES5 and ES6 versions of the language. The result are specific conclusions about the functional principles support in JavaScript, as well as a general conceptual framework that can be similarly used for analysing other functional programming languages and languages with functional features. We do not dive deeply into explaining the benefits of using these concepts, nor discussion of their appropriateness for various situations. This discussion may be found in the references provided.

3 Key Principles of Functional Programming

The formal foundation of functional programming is the Lambda calculus, also written as λ-calculus. It was formulated by mathematician Alonzo Church in the 1930 s as part of an investigation into the foundations of mathematics [5]. The Lambda calculus provides a simple semantics for computation, enabling properties of computation to be studied formally. We do not discuss these formal aspects of the Lambda calculus here, but we focus on the programming perspective.

The most fundamental principle is the notion of **first-class functions (P1)**. It is an essence of functional programming that functions are first-class citizens that may be manipulated as data. Since functions are considered values in their own right, it is natural for them to appear as arguments or results of other functions. Functions that takes other functions as arguments or that return functions as results are said to be **higher order**, and we refer to them as "functionals" to distinguish them from ordinary functions [6]. Functions may be named or *anonymous*, which corresponds to the notion of a Lambda function. Higher-order functions can be used for example in *functors*, where functionals are mapped over a structure, which provides a "better" alternative to iteration [7].

The **referential transparency (P2)** states that function call can be replaced by its return value (obtained by calling the function with the same arguments). It makes the order and count of execution irrelevant. This can be done with so-called *pure functions*. This means that function can not impose *side effects*, like a mathematical function. An example of a side effect is printing out some message, waiting for an input, sending something through the network, or only accessing some variable outside the function scope (for example global

variable or variable from parent scope). This poses severe limitations, however abiding the referential transparency has the following benefits [8]:

- Purely functions are remarkably easy to parallelize.
- Pure functions lead to a high degree of code encapsulation and reusability.
- They are easier to reason about.
- Pure functions are very easy to write unit tests for.

The referential transparency is tightly bound with **the immutability of variables and values (P3)**. This means, that if we assign a value to some variable, we cannot reassign it. This variable will hold this value until the program stops. From this perspective, variables are more like identifiers for data values, than slots for some changing data. In some languages (like Haskell: [9]), immutability is strictly embedded in the language. However, in languages with imperative features, the situation is more complicated and immutability must be explicitly managed. It can be achieved in several dimensions. The first step is disabling reassigning another value to an already assigned variable. This is usually achieved by a keyword (`const`, `val`, `final`, etc.). The effect is that we cannot assign a new value to this variable, but we still can change the value itself. For example, when we define immutable (or constant) variable named `user1` and value of this `user1` will be the user with name "George", we can not set `user1` to another user later, but we still can change the name of "George" to "John". Of course, this problem does not hold for values like integers or strings, just the compound types. The solution of this problem is to define immutable variables transitively. In this case, user object will have all properties defined as constants, so we can not change its name to "John". In case that user contains another compound object as its property, this object have to be also immutable (contains only immutable properties). This leads us to a question of standard libraries. To support immutability, types defined in libraries must be immutable and functions in these libraries must support operations with immutable structures. These functions have to return a new object with updated values instead of changing the existing object. For example, a sort function is not allowed to change array to sorted array, but it has to leave the original array untouched and return a new sorted array.

A **closure (P4)** is simply a pairing of a function with its environment: the bound variables that it can see [10]. It means, that function can access values from the enclosing block (lexical scope). More specifically, not from the context in which they are called, but from the context in which are defined. Closures are used for making the code more clear and readable. They also provide encapsulation (like private members in the traditional object-oriented programming).

A **recursion (P5)** is used in functional programming instead of loops. It is a situation when a function calls itself. There is a special type of recursion called tail-recursion. A tail-recursive clause is a recursive clause of the form

$$p :- q_1, .., q_n, p,$$

where $n \geq 0$, i.e. the last the last call in the body is a recursive call to itself. It is well known that tail-recursion can be replaced by iteration. This is because

there are no more calls after the tail-recursive one, which means that its binding environment can, with a bit of care, be discarded and the space reused [11]. Thus, if a compiler supports tail recursion optimisation, it solves the stack overflow risk of recursion in case of tail recursion.

3.1 Additional Properties

Apart from the fundamental principles, we identified the following additional principles. Some of them also appear in other paradigms, like object-oriented programming or logic programming.

Lazy evaluation (P6) is a situation when a value of an expression is not calculated in the moment of declaration, but it is delayed until needed; It may also happen that the value is not evaluated ever [9]. The lazy evaluation is typically bound to pure functions, as side effects complicate the situation by the possibility that they may not be evaluated [12]. Lazy evaluation does not make much difference for atomic values, apart from possible small optimisation. However, their importance is substantial for collections. Lazy collections (usually lists) are a very common pattern for solving problems in functional style. They bring a possibility to work with virtually infinite streams by evaluating just the portions that are accessed [13–15].

Currying (P7) is another important additional principle of functional programming. Currying is a technique of transforming a function of multiple arguments into evaluating a sequence of functions, each taking one argument [16]. In fact, currying is the default mechanism in the Lambda calculus, while multiple-argument functions are technically just a "syntactic sugar". The same is true for the Haskell programming language, while most of other languages default to multiple arguments and use currying mostly to achieve *partially applied functions*. It is a situation when we pass fewer arguments to a function that the function expects [10]. It is a powerful abstraction mechanism enabling creation of specialised versions of functions [10].

Pattern matching (P8) is a well-known concept not limited to the functional programming. Pattern matching provides the means to inspect and decompose nested data structures in a single statement [17]. Using this construct, a programmer can define different behaviour of a function based on distinct values and data structures without writing if-then-else constructs, which makes it a device of polymorphism.

Polymorphism (P9) [18] is a powerful abstraction principle, again not only limited to functional programming, it is also one of corner stones of object-oriented programming). *Polymorphic functions* are functions whose operands (actual parameters) can be of more than one single type. *Polymorphic types* are types whose operations are applicable to values of more than one single type [19].

To sum up, a proper functional programming language should implement the concept of functions as first class values, support closures, immutable variables and pure referential-transparent functions. Recursion is generally supported in all today's languages including the imperative ones, however tail recursion optimisation is a welcome asset from the implementation perspective.

4 FP Analysis of JavaScript

Let us now explain how the identified principles are embodied in the JavaScript language (JS), arguably the most important language of the web [20]. JavaScript is designed as a dynamically typed scripting language. The current widely used edition is the ECMAScript 5th Edition (ES5) [21] and a new standard ECMAScript 6th edition (ES6) is available [22]. ES6 contains more direct support for functional programming constructs, but it is not fully supported in the current web browsers (in November 2016). The current adoption status may be checked in [23]. Currently, the code in ES6 for browsers is usually translated into the ES5 code for compatibility reasons.

4.1 ES5

ES5 is a shortcut for EcmaScript 5 from 2009 JavaScript standard [21]. In JavaScript, functions are first-class objects and can be assigned to variables, passed as function arguments or returned as a function result. Anonymous (Lambda) functions are supported in form of:

```
function(arguments,...){ return ___;}
```

We may conclude that principle (P1): first-class functions is supported in JS. Functors are supported with arrays, but there is a small amount of standard functions based on them. They can be supported using libraries like Lodash [24] or Underscore [25].

Principle (P2): referential transparency is not guaranteed nor managed, as functions are not required to be pure and also (P3): immutability is not directly supported. There is no syntax for specifying immutable variables and no functions or objects from standard library that would provide support for immutability. So when we write:

```
var a = [5,2,4,3];
a.sort();
console.log(a); // [2, 3, 4, 5]
```

The value of variable *a* will be [2, 3, 4, 5]. So the function `sort` sorts the original array instead of returning the sorted array as its return value and letting the original array unchanged. (In fact, this function returns the sorted array as a return value, nevertheless it sorts the original array anyway.) The only way, how to declare a (local) variable is using the keyword `var`[1], and variables can be reassigned. For example, this is a valid code:

```
var a = 10;
a = 20;
console.log(a) // 20
```

[1] A variable can be also declared without the `var` keyword, which makes the variable global.

There is immutable support for properties in objects:

```
var obj = {};
Object.defineProperty(obj, 'key', {
  configurable: false,
  writable: false,
  value: 'some data'
});
```

When the `writable` attribute for a property is set to false, any attempt to change the value of the property fails [21].

Object `obj` now contains property `key` with value `some data`. This property can not be changed (`writable: false`) or deleted (`configurable: false`). The second option is freezing an existing object:

```
var obj = {key: 'value 1'}
console.log(obj.key) // value 1
obj.key = 'value 2'
console.log(obj.key) // value 2
Object.freeze(obj)
console.log(obj.key) // value 2
obj.key = 'value 3'
console.log(obj.key) // value 2
```

Because of standard functions which mutate data, most of the code in JS is not referential transparent – (P2) is not supported. There is a possibility to mitigate this situation by using libraries providing basic support for immutable structures and functions manipulating these structures (for example `immutable.js` [26]).

Functions are scope for variables. Objects may be created from JavaScript functions. Properties of these objects can be accessed using closures. Because these properties can be also changed, using closures breaks the referential transparency – If we invoke closure C, then change a variable that is used in this closure and then invoke C for the second time, the result of this closure may be different.

Nevertheless, closures (P4) are present and provide a powerful mechanism, which is leveraged in introducing the concept of modules, which is missing in the language. Modules are implemented in the AMD library ([27]) by enclosing the exported functions in another function, thus forming a closure and providing encapsulation.

Principle (P5): recursion may be used in JS, however tail call optimization is not present. Functions that recurse very deeply can fail by exhausting the return stack [20].

As for the additional principles, (P6): lazy values are not supported, (P7): currying is not supported in the language, but it can be achieved indirectly [28] or using libraries [29]). (P8): pattern matching is not present.

(P9): Polymorphism is supported by a *prototype inheritance* [30] in a rather object-oriented fashion. Polymorphism in JavaScript deserves a deeper explanation, which is out of scope of this paper.

To sum up, we can say, that ES5 supports first-class functions, but it lacks other properties of functional programming. Some of them can be simulated indirectly or using libraries. Functional style is not idiomatic in standard JavaScript, but a number of libraries and projects leveraging them seems to be rising.

4.2 ES6

ES6 is the new (2015) JavaScript standard and it is backward compatible. It brings several improvements, which are explained e.g. in [31]. Let us discuss the changes related to the functional principles here.

The first improvement is a more elegant syntax for anonymous (Lambda) functions: the //arrow functions//. The code:

```
evens.map(function (v) { return v + 1 })
```

can be shorten to

```
evens.map(v => v + 1)
```

So we may say that ES6 syntactically supports (P1) better than ES5. Also, by using arrow functions, one can achieve a more expressive closure syntax (P4). Also, ES6 changes scoping of `this` from function scope to lexical scope. This change allows a direct closure code – we do not have to save `this` reference as in ES5. Instead of

```
var self = this
this.nums.forEach(function (v) {
    if (v % 5 === 0)
        self.fives.push(v)
})
```

it is now possible to write [32]:

```
this.nums.forEach((v) => {
    if (v % 5 === 0)
        this.fives.push(v)
})
```

ES6 also improves the support for immutability. In the language, there are now two new keywords for creating variables. Keyword `let` creates mutable variable but scoped lexical (instead of the functional scoped `var` keyword). Using `const` one can create a constant. The ES6 `const` keyword is used to declare read-only variables, i.e. the variables whose value cannot be reassigned [31]. So (P3) is supported, but not required. Immutable variable means that nothing else can be assigned to this variable, but properties of this object still can be changed.

ES6 also adds *collections* and new functions for immutable operations: `find()`, `map()`, `reduce()` do not change the original collection. This strongly supports referential transparent code (P3).

A support for *tail call optimization* has been added, which facilitates the usability of recursion (P5). A tail position call must either release any transient internal resources associated with the currently executing function execution context before invoking the target function or reuse those resources in support of the target function [22].

ES6 supports *generators*, which are essentially lazy collections. One can define an iterable sequence with generator function and the control flow can be paused and resumed, in order to produce a sequence of values (either finite or infinite). But this is not enough to support laziness (P6). This provides only a type of lazy collection, but there is still missing lazy function invocation or lazy values.

Currying (P7) is still not part of the language, the situation remains the same as with ES5: it is achievable using libraries.

As for the additional principles, ES6 introduced the *destructuring assignment*:

```
var list = [ 1, 2, 3 ]
var [ a, , b ] = list
[ b, a ] = [ a, b ]
```

It is essentially (P8): pattern matching, but just in a limited fashion, as it can not be used to create polymorphic functions based on destructuring.

The model of (P9): polymorphism has not changed in ES6.

5 Conclusions

It may be an interesting observation that JavaScript in spite of its C-like syntax offers many fundamental functional programming principles in its heart and a lot can be achieved by libraries. Programmers may thus leverage a good portion of functional power and elegance in this popular language. ES6 is obviously a step towards better functional principles support. JavaScript is one of the languages, which is getting closer to purely functional languages. The following table summarizes concepts and their support in languages.

Property	ES5	ES6
First-class functions (P1)	+	++
Referential transparency (P2)	–	-
The immutability of variables and values (P3)	-[a]	+
Closures (P4)	+	++
Recursion (P5)	+	++
Lazy evaluation (P6)	–	-
Currying (P7)	-[a]	-[a]
Pattern matching (P8)	–	-
Polymorphism (P9)	+	+

[a] can be reached using libraries.

As for the future work, the presented conceptual framework may be used to analyse other programming languages from the functional programming perspective.

References

1. List of programming languages, page Version ID: 750948731, November 2016. https://en.wikipedia.org/w/index.php?title=List_of_programming_languages&oldid=750948731
2. Kedar, S.: Programming Paradigms and Methodology. Technical Publications, google-Books-ID: gvm9TPE96t4C, January 2008
3. Clojure. http://clojure.org/
4. Warburton, R.: Java 8 Lambdas: Pragmatic Functional Programming. O'Reilly Media, Inc., google-Books-ID: qKUdAwAAQBAJ, March 2014
5. Computer Science. PediaPress, google-Books-ID: Yte2cXVES9EC
6. Cousineau, G., Mauny, M.: The Functional Approach to Programming. Cambridge University Press, Cambridge, google-Books-ID: vccmAAAAQBAJ, October 1998
7. Hughes, J.: Why functional programming matters. Comput. J. **32**(2), 98–107 (1989)
8. Vander Hart, L., Sierra, S.: Practical Clojure, 1st edn. Apress, 232 pages, June 2010
9. Bird, R.: Thinking Functionally with Haskell. Cambridge University Press, Cambridge, google-Books-ID: B4RxBAAAQBAJ, October 2014
10. O'Sullivan, B., Goerzen, J., Stewart, D.B.: Real World Haskell: Code You Can Believe In. O'Reilly Media, Inc., google-Books-ID: nh0okI1a1sQC, November 2008
11. Jouannaud, J.-P.: Functional Programming Languages and Computer Architecture: Proceedings, Nancy, France, September 16–19. Springer, Heidelberg (1985)
12. Pickering, R., Eason, K.: Beginning F# 4.0. Apress, google-Books-ID: puQgDAAAQBAJ, May 2016
13. Koval, K.: Swift High Performance. Packt Publishing Ltd., VfioCwAAQBAJ google-Books-ID: VoCwAAQBAJ, November 2015
14. Ballou, K.: Learning Elixir. Packt Publishing Ltd., google-Books-ID:ogUcDAAAQBAJ, January 2016
15. Alexander, A.: Scala Cookbook: Recipes for Object-Oriented and Functional Programming. O'Reilly Media, Inc., google-Books-ID:BSo2AAAAQBAJ, August 2013
16. Borges, L.: Clojure Reactive Programming. Packt Publishing Ltd., google-Books-ID: 1tePBwAAQBAJ, March 2015
17. Geller, F., Hirschfeld, R., Bracha, G.: Pattern Matching for an Object-oriented and Dynamically Typed Programming Language. Universittsverlag Potsdam, google-Books-ID: 2gM_93yaz_kC (2010)
18. Hu, Z., Hughes, J., Wang, M.: How functional programming mattered. National Sci. Rev. **2**(3), 349–370 (2015)
19. Luca Cardelli, P.W.: On Understanding Types, Data Abstraction, and Polymorphism, December 1985. http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf
20. Crockford, D.: JavaScript: The Good Parts: The Good Parts. O'Reilly Media, Inc., google-Books-ID: PXa2bby0oQ0C, May 2008
21. ECMAScript Language Specification, December 2009. http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262

22. ECMAScript 2016 Language Specification, June 2016. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
23. ECMAScript 6 compatibility table, November 2016. http://kangax.github.io/compat-table/es6/
24. Lodash. https://lodash.com/
25. Underscore.js. http://underscorejs.org/
26. Immutable.js. https://facebook.github.io/immutable-js/
27. amdjs/amdjs-api. https://github.com/amdjs/amdjs-api
28. A Beginner's Guide to Currying in Functional JavaScript, October 2015. https://www.sitepoint.com/currying-in-functional-javascript/
29. Ramda Documentation. http://ramdajs.com/
30. Daggett, M.E.: Expert JavaScript. Apress, google-Books-ID: HpoQAwAAQBAJ, November 2013
31. Prusty, N.: Learning ECMAScript 6. Packt Publishing Ltd., google-Books-ID: 9O13CgAAQBAJ, August 2015
32. ECMAScript 6: New Features: Overview and Comparison. http://es6-features.org/#Lexicalthis