Proceedings of the 10th International Conference on Applied Informatics
Eger, Hungary, January 30–February 1, 2017. pp. 73–80
doi: 10.14794/ICAI.10.2017.73

Using generator functions in algorithmic visualizations

Zoltán Czirkos

Budapest University of Technology and Economics
Department of Electron Devices
czirkos@eet.bme.hu

Abstract

Electronic course materials can provide students a better learning experience compared to traditional paper-based presentation. This is especially true in the case of programming, where proper understanding of algorithms is based on understanding the computational processes they perform.

Web pages are particurarly suited for the development of these materials, because the usual text and image based multimedia content can be accompanied by automatically generated visualizations. As the web browser platform is itself programmable, the visualizations shown by the browser can be controlled by the learner, and can even be interactive.

However, the web browser platform is inherently event-driven. That requires the developer to reformulate the algorithms to be presented in an unusual manner, usually by converting the structured program code to a finite-state automaton. This is time-consuming and error-prone work.

This article presents and evaluates several possible solutions to this problem. The main focus of the investigation is generator functions, a new feature of JavaScript, available in ECMA-262 since 2015. This new language feature enables the developer to implement the algorithms to be presented without any modification to their control structure, thereby making development and maintenance of the course materials easier.

Keywords: algorithm visualization, generator functions, javascript

MSC: 97Q60, 68N15, 68N20

1. Introduction

The key to understanding imperative programming is understanding the temporal nature of the execution of algorithms. Variables in a program change their value, therefore expressions are evaluated to different values as well [4]. Students have

```
let max = arr[0];
for (let i = 1; i != n; ++i)
    if (arr[i] > max)
        max = arr[i];
```

Figure 1: Maximum selection

to understand how data is manipulated over time, because variables' names might not always represent their actual content.

Consider the well known algorithm of maximum selection in Figure 1. In this code, the expression `arr[i] > max` does not make any sense on its own, without context. "No element in the array is greater than the maximum element of the array", one having no experience with imperative programming might argue. However, the variable `max` is not necessarily storing the maximum value while the loop is running; the postcondition $\forall i, max \geq arr[i]$ only holds after completing all the iterations.

The temporal nature of programs calls for visualizations. When referring to algorithm visualizations, we usually mean *visualizing the data* [7], and not the algorithms themselves. The control flow of the algorithm is static, and cannot be animated. Also, a flow chart does not help either the understanding of algorithms, nor does it teach students how to come up with new ones. However by seeing data being manipulated, one can understand the intention behind the lines of the code.

Modern browsers enable us to develop e-learning materials which provide students a highly interactive learning environment. As opposed to books and static slide presentations, this medium can contain animations which, unlike video snippets, can even be interactive. With proper stylesheets, web pages can be used as presentations as well. Research shows that this kind of technology is rather underdeveloped [1]. That is a surprising fact, as lecturers who teach programming have the ability to develop technology to implement these animations easily – maybe create these animations automatically.

Please note that the improvement of learning effectiveness is not considered in this article (see [6] for an in-depth analysis of motivational aspects). The aim of this research was to develop a framework, which enables one to create visualizations of algorithms in a browser. The main goal is that coding the animation should be as simple as possible, specifically: the code generating the animation should have the same control structure as the algorithm to be presented to the students.

2. Preserving state in an asynchronous environment

The asynchronous, event-driven nature of web applications is problematic with respect to the constraint of the animation algorithm having the same control structure as the algorithm that it presents.

Consider the simple loop on Figure 2 as an example. The task is to create a

```
let i = 1;

while (i <= 10) {
    print(i);

    i += 1;

}
```

(a) The algorithm being explained

```
let i = 1;
wait_for_click();
while (i <= 10) {
    print(i);
    wait_for_click();
    i += 1;
    wait_for_click();
}
```

(b) Desired control structure

Figure 2: The algorithm controlling the animation.

debugging-like behavior, where the loop is paused before each iteration, in order to explain to the students how the value of the loop variable is compared to the limit.

Figure 2a is the algorithm being presented. Figure 2b shows how the source code behind the slide should ideally look like. It should have an identical control structure, with some "wait here for the next mouse click" instructions inserted, which enable the instructor to explain what is happening.

However, in an event-driven environment, the control is inverted. One does not simply call a function to wait for the click of a button, but rather a callback function is bound to the mouse click event. The loop of the source code cannot be inside the body of the click handler, because it would restart counting on each invocation of the function, ie. on each mouse click. Also we must not return from inside the loop, otherwise the state, the value of the `i` variable would be lost.

Note that by referring to the state of the loop, we also refer to information which is only stored in the execution environment: the instruction that is currently executed. This state cannot be accessed in any way from the code (at least not in a language which does not support continuations [2]). If we break the loop, we lose the state. Any attempt to implement the wait for the next click will fail, as the click handler must return, and cannot call the above code snippet as a function either.

There are several partial solutions to the problem, but with conventional elements of structured programming, each has its own disadvantages.

2.1. Converting the algorithms to finite automata

Any algorithm can be represented as a finite automaton, ie. a state machine. In this application, each mouse click would instruct the state machine to execute the next set of instructions and make the transition to the next state. The states and transitions of the automaton would be equivalent to the control flow graph of the algorithm, with extra states added where a mouse click is expected.

This automaton might be coded in different ways. One approach is to implement a class which represents the state machine, with its attributes being the local

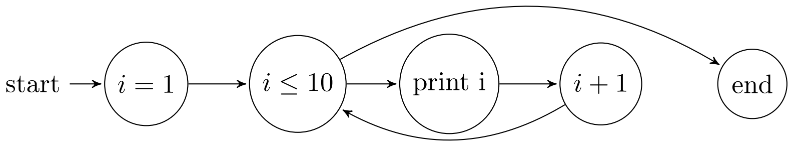

Figure 3: State machine model of the algorithm on Figure 2b (illustration only)

```
let events = [];
let i = 1;
while (i <= 10) {
    events.push(function(i) {
        print(i);
    }.bind(null, i));
    i += 1;
}
```

(a) Recording animation events

```
let step = 0;
button.addEventListener('click',
  function() {
    if (step < events.length) {
        events[step]();
        step += 1;
    }
  });
```

(b) Playback

Figure 4: Using closures to store visual effects of the animation

variables of the original code and the state variable itself.

The problem with this approach is that it is tedious and error-prone work for the author to convert the algorithms to state machines. Figure 3 shows a simple loop. Any other algorithm would get quite complex, the number of states grows with the cyclomatic complexity of the algorithm and the required timing stops of the animation as well. Automating this would require implementing a compiler, which reads JavaScript code to translate the algorithm into a state machine representation.

2.2. Recording events of the animation

An elegant but incomplete solution is the use of closures to store visual effects of the animation.

A closure is a function which captures its environment when created, and can use data from that environment when executed later. In JavaScript, closures are first class citizens of the language, and can be stored in variables like any other regular value [3]. Figure 4 shows an implementation of this idea: an array of closures represents the steps on the animation and the functions are called later, one at a time.

The idea is similar to recording the steps of the animation as a video, and then playing it back frame by frame. This has a clear advantage compared to the previous solution regarding the complexity of the algorithm: the control structure is the same as the original, it is a simple loop. It could be easily adapted to recursive

```
function* infinite_fibonacci() {
    let current = 0, next = 1;
    while (true) {
        yield current;
        let temp = current + next;
        current = next;
        next = temp;
    }
}
```

(a) Definition of a generator function

```
let gen = infinite_fibonacci();
for (let i = 1; i <= 10; ++i)
    print(gen.next().value);
```

(b) Instantiation of the generator

Figure 5: Generator function yielding an infinite stream of Fibonacci numbers

functions as well.

The disadvantage however is that all interactivity is lost. If the algorithm to be presented requires some input while it is running (for example reading the limit of the loop from the keyboard), its output cannot be determined in advance.

2.3. Using an interpreter

Storing the state of a running algorithm, pausing and resuming it, is also possible by creating a full-fledged interpreter. In this case, the program code controlling the animation would be written in a new, domain-specific programming language designed specifically for this task. The JavaScript application in the browser would be running that code, thereby having complete control over its execution.

This approach requires tremendous work. Defining a new language, implementing its parser and execution framework is a complex project. There are open source projects aiming for this task [5], but these interpreters would have to be extended for this application. The most notable difficulty is that the program running inside the interpreter should be provided with access to the running environment itself. For example, the animation might contain some statements to print a number to the screen or to set the color of a shape in a figure – this is only possible if it has access to the DOM (Document Object Model) of the web page it is embedded into.

3. Using generator functions as animation timers

The 2016 version of JavaScript defines the notion of generators [3]. A generator function is a special function capable of yielding many values. Instead of having `return` statements, these use the keyword `yield` to “send” a value to their caller. Contrary to normal returning, yielding a value does not stop the execution of the function, rather it is only suspended. Therefore its state, local variables are

```
function* simple_loop() {
    let i = 1;
    yield false;
    while (i <= 10) {
        print(i);
        yield false;
        i += 1;
        yield false;
    }
    yield true;
}
```

(a) Loop animation. Execution will be paused at the yield statements

```
let anim1 = simple_loop();
button.addEventListener('click',
  function() {
    anim1.next();
  });
```

```
let anim2 = simple_loop();
let timer = null;
function next() {
    if (gen.next().value)
        clearInterval(timer);
}
timer = setInterval(next, 350);
```

(b) Running the animation. Step by step execution with user interaction and timed execution

Figure 6: Values yielded can be used to control the animation

preserved in the running environment, and execution can be resumed later. Figure 5 shows a generator function that yields an infinite stream of Fibonacci numbers.

The `yield` statement can be used to suspend an algorithm and resume it later. In the animation application, the animation algorithm itself will be a generator function. At each state when the animation is to be stopped to wait for a click, a `yield` statement must be inserted into the code. This will pause the execution of the algorithm until it is resumed by the following `next` member function call of the instantiated generator. Figure 6a shows how this can be implemented in JavaScript. Note that the animation is not prerecorded, so it can be interactive as well – it can even input data from the user while it is running.

3.1. Utilizing the values yielded to control the animation

Generators are able to send values to their caller via the `yield` statement. In our case, this can be used as a communication channel between the algorithm and the controlling environment. For example, yielding the value `false` might represent that the algorithm is still running, and `true` might tell the controller that the animation is finished, ie. there are no more steps. In Figure 6b this this is used to stop the timer controlling the animation.

JavaScript also has a `yield*` keyword, which instantiates a generator, and calls its `next()` method until the generator is finished. Meanwhile it yields all the values the generator has yielded. This enables the author to use recursion in the animations. Figure 7 is and example of a tree traversal routine written in this style.

Using `yield*` in the recursive call creates a behavior that is well known as “step into” in debuggers of integrated development environments. As all the values are

```
function* traverse_tree(root) {
    if (root != null) {
        yield* traverse_tree(root.left);
        yield;
        yield* traverse_tree(root.right);
    }
}

yield* traverse_tree(root);
```

Figure 7: Animating recursive functions

passed to the caller, the animation controller will wait until the next click for each node of the tree. To create a "step over" like behavior (for which the function call is executed completely, not step by step), one could simply instantiate the generator and call `next()` on it as many times as necessary, but without yielding anything.

4. Conclusion

Generator functions can be used very effectively to implement algorithmic visualizations in JavaScript. The ability to suspend and resume execution of a function allows one to implement the visualization with the same control structure as the algorithm itself to be visualized. Also the algorithm is running directly inside the browser, thereby it has access to the DOM. Therefore development of interactive course material in a web page is simple and straightforward with this technique.

References

[1] Shaffer, Clifford A., et al. Algorithm visualization: The state of the field. *ACM Transactions on Computing Education (TOCE)* 10.3 (2010): 9.

[2] Reynolds, John C., The discoveries of continuations. Lisp and symbolic computation 6.3 (1993): 233-247.

[3] Ecma International, ECMAScript ® 2016 Language Specification. `https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`. Retrieved 5th Dec. 2016.

[4] Dehnadi, Saeed, Richard Bornat, and Ray Adams. Meta-analysis of the effect of consistency on success in early learning of programming. PPIG, 2009.

[5] JSCPP, A simple C++ interpreter written in JavaScript. `https://github.com/felixhao28/JSCPP`. Retrieved 12th Apr 2017.

[6] Nikula, Uolevi, Orlena Gotel, and Jussi Kasurinen. A motivation guided holistic rehabilitation of the first programming course. ACM Transactions on Computing Education (TOCE) 11.4 (2011): 24.

[7] Victor, Bret. Learnable programming. Worrydream.com (2012). `http://worrydream.com/LearnableProgramming/`. Retrieved 19th Apr 2017.