Rethinking JavaScript Loops as Combinators

# Prashant Singh, Student, MPSTME, Mumbai, [Pr](mailto:Prashantsingh109@ymail.com)[ashantsingh109@ymail.com](mailto:ashantsingh109@ymail.com) Rejo Mathew, Asst Prof, Dept. of IT, MPSTME, Mumbai, [rejo.mathew@nmims.edu](mailto:rejo.mathew@nmims.edu) Veerdhwaj Singh, Student, MPSTME, Mumbai, [veerdhw](mailto:veerdhwaj@gmail.com)[aj@gmail.com](mailto:aj@gmail.com)

**Abstract**—Loop combinators like *map* and *filter* have fallen out of use in performance critical sections of code in favor of *for* and *while* loops. This is tragic because not only are loop combinators more expressive than loop control structures but also they are simpler and more error resistant. Using inductive loop combinators, performance can be improved without sacrificing expressiveness. The resulting code is more modular and easier to maintain.

**Index Terms**—Functional Programming, JavaScript, Loop Combinators, Stateful Combinators, Method Chaining.

# ◆

1. **INTRODUCTION**

OOPS are one of the cornerstones of programming. Without loops every program would be uninter- esting. Since loops are so important, it is paramount that they are efficient. Traditional loop control struc- tures like *for* and *while* allow programmers to write ef- ficient iteration constructs. However, they are neither

L

modular nor composable.

Modularity and composability are the key to suc- cessful programming [1]. Together, they help combat software complexity — the biggest contributor to the “software crisis” identified in 1968 [2]. Functional programming provides new ways to decompose and combine programs, thereby mitigating software com- plexity [3].

In functional programming languages, loop combi- nators like *map* and *filter* are used instead of traditional loop control structures. Loop combinators are more expressive than loop control structures, but they are less efficient. For example, consider the following code which filters all the odd numbers in a list and then increments them by one:

***var* result= [1,2,3,4,5].filter(odd).map(inc);**

Here, we are running two separate loops instead of one. Hence, the above functional program is less efficient than its equivalent imperative program:

***var* a= [1,2,3,4,5];**

***var* l=a.length; *var* i= 0,j= 0; *var* result= [];**

***while* (i<l) {**

***var* n=a[i++];**

***if* (odd(n)) {**

**result[j++] =inc(n);**

**}**

**}**

The question is how to make loop combinators more efficient so that we don’t have to write loop control structures in performance critical sections of code? There are several existing solutions to this prob- lem in JavaScript, inspired by transducers in Clojure [4]. However, we provide a solution that is on average thrice as fast as existing solutions:

* We define precisely what a loop combinator is and explain how they are composed (Section 2).
* We describe an alternative method of compos- ing loop combinators using method chaining (Section 3).
* We describe how to create stateful combinators in Section 4, and how to exit from the loop early in Section 5.
* We describe an array specific optimization technique in Section 6, comparing the perfor- mance of our solution with existing solutions in Section 7.
* We conclude by showing how using loop combinators helps combat software complexity (Section 8).

1. **BACKGROUND**

In this section we lay the groundwork for the rest of the paper by providing a mathematical model of a loop. Furthermore, we use this model to give a precise definition of a loop combinator. To understand what a loop is, let’s take an example of a simple loop which computes the sum of a list of numbers:

978-1-4673-7309-8/15/$31.00 ©2015 IEEE 288

***var* a= [1,2,3,4,5];**

***var* i= 0,l=a.length;**

***var* sum= 0;**

***while* (i<l) {**

**sum+=a[i++];**

**}**

This loop consists of three distinct elements: the initial result of the loop, the body of the loop which updates the result on every iteration and an input tape which produces one input element for every iteration of the loop body. The loop body may be executed several times. However, the loop always has an associated result even if the loop body is never executed.

For the purpose of abstraction, we won’t concern ourselves as to how the input tape produces an input element for the loop body. The input tape could be a list which produces its own elements as input ele- ments, or it could be a natural number which produces its predecessors as input elements, etc.

Let’s assume that *A* is the smallest set that contains all the input elements produced by the input tape and

that *B* is the set of all the possible results of the loop. Then the loop is defined by the 4-tuple *A, B, B*0*, F* where *B*0 *B* is the initial result of the loop and

*× →*

*∈*

*( )*

*F* : *B A B* represents the loop body.

Since, the loop is decoupled from the input tape

we can use the same loop with any kind of input tape.

defined by the 4-tuple N*,* N*,* 0*,* + . Hence, it can be For example, the aforementioned summation loop is used with any kind of input tape that produces natural

*( )*

numbers.

Now that we provided a mathematical model for loops we can proceed to give a precise definition for loop combinators. A loop combinator is a function that transforms an input tape. For example, *map* and *filter* are loop combinators that transform lists. Loop combinators can be combined to form more complex loop combinators.

However, loop combinators need to be indepen- dent of the input tape. Since the loop body already ab- stracts away the input tape, an input tape independent loop combinator is simply a loop body transformer. For example, here are the definitions of the input tape independent *map* and *filter* loop combinators:

***function* map(functor) {**

***return function* (body) {**

***return function* (result,value) {**

***return* body(result,functor(value))**

**;**

**body(result,value) : result;**

**};**

**};**

**}**

A loop body is a partial function *F* : (*result* :

*B*) (*value* : *A*) *B*. A loop combinator is a partial function *G* : *R*((*R B R*) *R A R*). Notice that the loop combinator knows nothing about the

*∀ × → → × →*

*× →*

result of the loop. However, by returning the previous result instead of calling the output loop body it can continue with the next iteration. In Section 5 we define an alternative semantics of continuation.

Loop combinators can be composed using simple function composition. For example, the loop that com- putes the sum of all the odd numbers incremented

by one can be defined as N*,* N*,* 0*,* (*filter*(*odd*) *map*(*inc*))(+) . This reads from left to right as “filter all the odd numbers, increment them by one and then

*)*

*( ◦*

compute their sum.”

1. **METHOD CHAINING**

Object-oriented languages like JavaScript do not have a function composition operator. Although defining function composition is simple yet it’s only efficient when composing two functions. When composing more than two functions, we create unnecessary in- termediate functions:

***function* compose(f, g) {**

***return function* (x) {**

***return* f(g(x));**

**};**

**}**

Given, (*f* : *c → d*), (*g* : *b → c*) and (*h* : *a → b*), we get (*e* : *a → d*) as follows:

***var* e = compose(compose(f, g), h); // or**

***var* e = compose(f, compose(g, h));**

termediate functions (*f g*) and (*g h*) is to create One way of getting rid of these unnecessary in- a generic function, *chain*, to compose a chain of func-

*◦ ◦*

tions:

***var* e = chain([f, g, h]);**

***function* chain(xs) {**

***return function* (x) {**

***var* fs = xs, length = fs.length;**

***while* (length > 0) x = fs[--length](x);**

***return* x;**

**};**

**}**

**}; };** However, a more object-oriented way would be

**}** to use method chaining. It is widely known that

*·*

*· · ·*

*◦ ◦*

***function* filter(predicate) {**

***return function* (body) {**

***return function* (result,value) {**

***return* predicate(value) ?**

(*f g h*)(*x*) in functional languages is equivalent to *x h*() *g*() *f* () in object-oriented languages where *x* is an object, ( ) is the membership operator and *h*(), *g*() and *f* () are method calls. Hence, we can

write (*filter*(*odd*) *◦ map*(*inc*))(+) in JavaScript as

*chain*(+) *· map*(*inc*) *· filter*(*odd*) *· value*(): **}; }**

***var* LoopBody = defclass({**

**body(result,value);**

**constructor: *function* (body) { this.body = body;**

**},**

**map: *function* (functor) {**

***var* body = this.body;**

***return new* LoopBody(mapBody);**

***function* mapBody(result, value) {**

***return* body(result, functor(value))**

**;**

**}**

**},**

**filter: *function* (predicate) {**

***var* body = this.body;**

***return new* LoopBody(filterBody);**

***function* filterBody(result, value) {**

***return* predicate(value) ? body(result, value) :**

However, this will only work once after which it will fail. Thus, this is not a good solution because it is not reusable. One way to make stateful loop combinators reusable is to create them anew for each loop. Indeed, this is what transducers in Clojure do [6]. However, creating a new chain of combinators for every loop is inefficient.

As an alternative, we propose to refresh the state once for every loop using an *init* function. In Section 6, we’ll extend this function to implement an array spe- cific optimization technique. The *init* function belongs to the same scope as the loop body and hence, it can modify the state of the loop body.

Although the *init* function is only used for its side effect yet it’s still a function. Hence, the question arises as to what is its domain and codomain. Since we

don’t care about either of them, we get the partial

**}**

**}**

**});**

**result;**

function *init* : *A*(*A A*). Hence, the *init* chain of functions always returns its argument whether or not

it performs any side effects.

*∀ →*

***var* body = *new* LoopBody(add)**

**.map(inc)**

**.filter(odd)**

**.body;**

**// (1 + 1) + (3 + 1) + (5 + 1) = 12**

**console.log([1,2,3,4,5].reduce(body, 0));**

***function* add(x, y) { *return* x + y; } *function* inc(x) { *return* x + 1; } *function* odd(x) { *return* x % 2 === 1; }**

***function* defclass(prototype) {**

***var* constructor = prototype.constructor; constructor.prototype = prototype; *return* constructor;**

**}**

This is commonly known as the chain-value pat- tern in JavaScript, which rose to prominence because of the popular Underscore.js library [5].

1. **STATEFUL COMBINATORS**

***var* LoopBody = defclass({**

**constructor: *function* (init, body) { this.init = init;**

**this.body = body;**

**},**

**map: *function* (functor) {**

***var* body = this.body;**

***return new* LoopBody(this.init, mapBody)**

**;**

***function* mapBody(result, value) {**

***return* body(result, functor(value))**

**;**

**}**

**},**

**filter: *function* (predicate) {**

***var* body = this.body;**

***return new* LoopBody(this.init, filterBody);**

***function* filterBody(result, value) {**

***return* predicate(value) ? body(result, value) :**

**result;**

Until now, we’ve only created stateless loop combi- **}**

nators like *map* and *filter*. However, there are many **},**

useful loop combinators that require state such as the *drop* combinator which skips the first *n* elements of an input tape. The na¨ıve solution is add a stateful loop combinator to the prototype of *LoopBody*:

**LoopBody.prototype.drop= *function* (n) {**

***var* body=this.body;**

***return new* LoopBody(dropBody);**

***function* dropBody(result,value) {**

***return* n> 0 ?**

**(n--,result) :**

**drop: *function* (\_n) {**

***var* n;**

***var* init = this.init;**

***var* body = this.body;**

***return new* LoopBody(dropInit, dropBody)**

**;**

***function* dropInit(x) { n = \_n;**

***return* init(x);**

**}**

**}**

**});**

***function* dropBody(result,value) {**

***return* n> 0 ?**

**(n--,result) : body(result,value);**

**}**

Since our definition of a loop doesn’t capture the semantics of *break* we need to revise our definition. This means that we won’t be able to execute our loops with a simple *reduce* function like we’ve been doing all along. We will need something more powerful. In Clojure the *reduce* function can be explicitly terminated

***var* loop= *new* LoopBody(id,add)**

**.map(inc)**

**.filter(odd)**

**.drop(5);**

**loop.init();**

**// (7 + 1) + (9 + 1) = 18 console.log([1,2,3,4,5,6,7,8,9,10]**

**.reduce(loop.body, 0));**

***function* id(x) { *return* x; } *function* add(x,y) { *return* x+y; } *function* inc(x) { *return* x+ 1; }**

***function* odd(x) { *return* x% 2 === 1; }**

***function* defclass(prototype) {**

***var* constructor=prototype.constructor; constructor.prototype=prototype; *return* constructor;**

**}**

Notice that stateless loop combinators like *map* and *filter* don’t have their own *init* functions. They simply return the previous function. This makes refreshing the state a little more efficient than creating the entire loop combinator chain anew. However, there’s also a downside to this. You can’t use the same loop body concurrently because state would be shared. However, in practice this only happens in asynchronous loops and can be easily avoided.

1. **EARLY TERMINATION**

Traditional loop bodies have a certain amount of con- trol over the flow of the loop itself. You can skip the rest of the iteration using *continue* and you can break out of the loop when you get the final answer without having to read the rest of the input tape using *break*. In fact, our current definition of a loop body captures the semantics of *continue*.

The loop body is a partial function *F* : *B A*

*→ →*

*B* where *B* is the set of all the possible results of

the loop and *A* is the smallest set that contains all the input elements produced by the input tape. By returning the previous result, a loop combinator can effectively skip the rest of iteration. In fact, this is precisely what the *filter* and *drop* combinators do.

Unfortunately, our current definition of a loop

body does not capture the semantics of *break*. Hence,

by applying *reduced* to the output of the loop body [7]. Although the semantics of *break* is explicit in Clo- jure yet the semantics of *continue* is implicit. However, we believe, in accordance with the Zen of Python, that explicit is better than implicit [8]. Therefore, in our new definition of a loop body we made both *break* and

*continue* explicit *F* : *B A B, bool* .

*→ → ( )*

The set *bool* is a set of two elements, *true* and

*false*. Hence, we may decide whether *break* corre- sponds to *true* or *false*. Although it doesn’t really matter yet to follow a convention we choose *true* to denote *break* and *false* to denote *continue*. We also define a special *reduce* function for arrays which allows early termination:

***function* reduce(loop,a,xs) { loop.init();**

***var* f=loop.body; *var* l=xs.length; *var* i= 0;**

***while* (i<l) {**

***var* pair=f(a,xs[i++]);**

***if* (pair.snd) *return* pair.fst; a=pair.fst;**

**}**

***return* a;**

**}**

Now that we’ve defined the semantics of both *break* and *continue* we can define the *take* loop combinator. In addition, we must also redefine the previous loop bodies so that they explicitly *break* or *continue*. Since the *map* loop combinator doesn’t *break* nor *continue*, it doesn’t need to be modified.

***var* LoopBody = defclass({**

**constructor: *function* (init,body) { this.init=init;**

**this.body=body;**

**},**

**map: *function* (functor) {**

***var* body=this.body;**

***return new* LoopBody(this.init,mapBody)**

**;**

***function* mapBody(result,value) {**

***return* body(result,functor(value))**

**;**

we can’t define certain loop combinators like *take*, **}, }**

which takes the first *n* elements of an input tape and skips the rest. Although we could keep skipping the rest of the elements using the semantics of *continue* yet that is neither efficient nor would it work on infinite input tapes.

**filter: *function* (predicate) {**

***var* body=this.body;**

***return new* LoopBody(this.init, filterBody);**

***function* filterBody(result,value) {**

***return* predicate(value) ? body(result,value) :**

**{fst:result,snd: *false* };**

**}**

**},**

**drop: *function* (\_n) {**

***var* n;**

***var* init=this.init;**

***var* body=this.body;**

***return new* LoopBody(dropInit,dropBody)**

**;**

***function* dropInit(x) { n=\_n;**

***return* init(x);**

**}**

***function* dropBody(result,value) {**

***return* n> 0 ?**

**(n--, {fst:result,snd: *false***

**}) :**

**body(result,value);**

**}**

**},**

**take: *function* (\_n) {**

***var* n;**

***var* init=this.init;**

***var* body=this.body;**

***return new* LoopBody(takeInit,takeBody)**

**;**

***function* takeInit(x) { n=\_n;**

***return* init(x);**

**}**

***function* takeBody(result,value) {**

***return* n> 0 ?**

**(n--,body(result,value)) :**

**{fst:result,snd: *true* };**

**}**

**}**

**});**

***var* loop= *new* LoopBody(id,add)**

**.map(inc)**

**.filter(odd)**

**.take(5);**

**// (1 + 1) + (3 + 1) + (5 + 1) = 12**

**console.log(reduce(loop, 0,**

**[1,2,3,4,5,6,7,8,9,10]));**

***function* id(x) { *return* x; }**

***function* add(x,y) { *return* {fst:x+y,**

**snd: *false* }; }**

***function* inc(x) { *return* x+ 1; }**

***function* odd(x) { *return* x% 2 === 1; }**

***function* defclass(prototype) {**

***var* constructor=prototype.constructor; constructor.prototype=prototype; *return* constructor;**

**}**

1. **OPTIMIZATION FOR ARRAYS**

Until now, we’ve only used the summation loop. An- other common kind of loop is a generative loop. For example, we can create a loop that generates an array. The traditional way to do this would be to create an empty array for every loop and then push elements to the end of the array.

***var* toArray= *new* LoopBody(id,push);**

**// a new array [1,2,3,4,5] console.log(reduce(toArray, [], [1,2,3,4,5]));**

***function* push(result,value) { result.push(value);**

***return* {fst:result,snd: *false* };**

**}**

However, if we knew the length of the array be- forehand then we could allocate just enough space for the array, thereby making the loop faster. The question is, how do we determine the length of the array beforehand? As it turns out, the *init* function which we’ve only been using for its side effect of refreshing the state is the perfect solution to this problem.

Currently, we have defined *init* : *A*(*A A*)

*∀ →*

change its type to *init* : N *R* where *R* is the set of all which is not very useful as a function. However, if we the possible results of the loop then we can return the

*→*

initial result of the loop, given the minimum expected number of loop iterations, in addition to refreshing the state of the loop.

This would mean that the entire loop would be represented as a single data structure. However, some stateless loop combinators like *filter* would also have to implement the *init* function. Nevertheless, the per- formance gain for generating arrays would make it worth implementing. Finally, we would have to mod- ify the definition of our special *reduce* function for arrays.

***function* reduce(loop, xs) {**

***var* l = xs.length; *var* a = loop.init(l); *var* f = loop.body; *var* i= 0;**

***while* (i<l) {**

***var* pair=f(a,xs[i++]);**

***if* (pair.snd) *return* pair.fst; a=pair.fst;**

**}**

***return* a;**

**}**

***var* Loop=defclass({**

**constructor: *function* (init,body) { this.init=init;**

**this.body=body;**

**},**

**map: *function* (functor) {**

***var* body=this.body;**

***return new* Loop(this.init,mapBody);**

***function* mapBody(result,value) {**

***return* body(result,functor(value))**

**;**

***function* toArrayInit(length) { i= 0;**

***return* length> 0 ?**

***new* Array(length) : [];**

**}**

**}**

**}, *function* toArrayBody(result,value) {**

**filter: *function* (predicate) {**

***var* init=this.init;**

***var* body=this.body;**

***return new* Loop(filterInit,filterBody)**

**}**

**}());**

**result[i++] =value;**

***return* {fst:result,snd: *false* };**

**;**

***function* filterInit(length) {**

***return* init(0);**

**}**

***function* filterBody(result,value) {**

***return* predicate(value) ? body(result,value) :**

**{fst:result,snd: *false* };**

**}**

**},**

**drop: *function* (\_n) {**

***var* n;**

***var* init=this.init;**

***var* body=this.body;**

***return new* Loop(dropInit,dropBody);**

***function* dropInit(length) { n=\_n;**

***return* init(Math.max(length-n, 0)**

**);**

**}**

***function* dropBody(result,value) {**

***return* n> 0 ?**

**(n--, {fst:result,snd: *false***

**}) :**

**body(result,value);**

**}**

**},**

**take: *function* (\_n) {**

***var* n;**

***var* init=this.init;**

***var* body=this.body;**

***return new* Loop(takeInit,takeBody);**

***function* takeInit(length) { n=\_n;**

***return* init(Math.min(length,n));**

**}**

***function* takeBody(result,value) {**

***return* n> 0 ?**

**(n--,body(result,value)) :**

***var* loop=toArray**

**.map(inc)**

**.filter(odd)**

**.take(5)**

**.drop(2);**

**// prints [4, 6, 8] console.log(reduce(loop,**

**[1,2,3,4,5,6,7,8,9,10]));**

***function* inc(x) { *return* x+ 1; }**

***function* odd(x) { *return* x% 2 === 1; }**

***function* defclass(prototype) {**

***var* constructor=prototype.constructor; constructor.prototype=prototype; *return* constructor;**

**}**

It’s interesting to note that *dropInit* calls the next *init* function with the maxiumum of the *length n* and zero while *takeInit* calls the next *init* function with the minimum of the *length* and *n*. It’s even more inter- esting to note that *filterInit* calls the next *init* function with zero. This is because *filter* can’t determine in advance how many elements it will filter. Hence, it returns the minimum number of possible elements (i.e. zero). Thus if you have a *filter* in your chain then this optimization won’t work.

*−*

1. **RELATED WORK AND BENCHMARKS**

The loop combinators that we described in this paper are very similar to the Rich Hickey’s transducers in Clojure [4]. In fact, our work on loop combinators has been inspired by Clojure transducers. However, there are several important differences between our loop combinators and Clojure transducers.

First, our loop combinators aren’t as powerful as Clojure transducers because we don’t have an output function that ties up loose ends after the loop body has completed its execution. This means that a few loop combinators, such as the *partition-by* combinator in

**}**

**}**

**});**

**{fst:result,snd: *true* };**

Clojure, cannot be implemented. Nevertheless, adding the output function to the definition of the loop is straightforward.

***var* toArray= (*function* () {**

***var* i;**

***return new* Loop(toArrayInit,toArrayBody);**

Reducers in Clojure are just Moore machines [9] and transducers in Clojure are just Mealy machines

[10] augmented with the ability to perform entry actions. Hence, our loop combinators are strictly less powerful than Moore machines and Mealy machines.

However, augmented with an output function they would be just as powerful as transducers in Clojure.

That being said, we are not the first people to import Clojure transducers to JavaScript. There are a lot of implementations of transducers in JavaScript. However, we believe that our approach to transducers have a lot of novel ideas that could be beneficial for everyone. We benchmarked our code against Lo- Dash, the fastest functional programming library in JavaScript, and the results look quite promising.

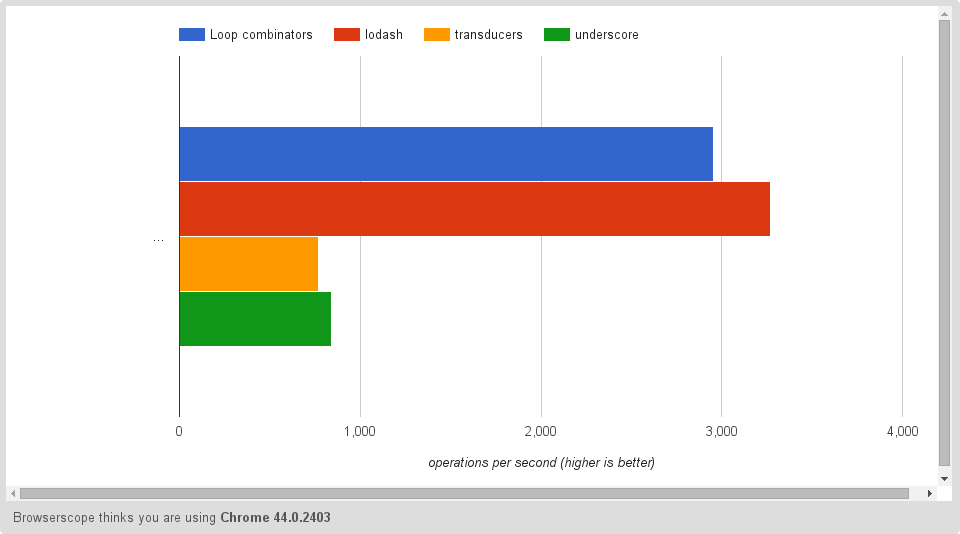


Fig. 1. Comparing *toArray map*(*inc*) *filter*(*odd*) across li- braries.

*· ·*

1. **CONCLUSION**

Our goal was to promote the use of loop combinators over traditional loop control structures in order to reduce software complexity by making loop combi- nators more efficient. Although our loop combinators are quite efficient yet they are slower than the fastest combinator libraries. However, in the process of cre- ating this library we discovered some insightful facts about method chaining and loop optimization which we hope will be helpful to others.

**ACKNOWLEDGMENTS**

The authors would like to thank Adit Shah for his im- mensely helpful insight into loop combinators, with- out which this paper wouldn’t be.

**REFERENCES**

1. B. Victor, “Learnable programming,” [http://worrydream.com/.](http://worrydream.com/)
2. B. Moseley and P. Marks, “Out of the tar pit,” *Software Practice Advancement (SPA)*, 2006.
3. J. Hughes, “Why functional programming matters,” *The computer journal*, vol. 32, no. 2, pp. 98–107, 1989.
4. R. Hickey, “Transducers,” Strange Loop, 2014.
5. J. Ashkenas, “0.4.0 is out, with oop-style and chaining,” [http://underscorejs.org/#chaining,](http://underscorejs.org/#chaining) 2009.
6. R. Hickey, “Inside transducers,” Clojure/conj, 2014.
7. R. Hickey and S. Davis, “reduced - clojure.core — clojure- docs,” https://clojuredocs.org/clojure.core/reduced, 2013.
8. T. Peters, “The zen of python,” in *Pro Python*. Springer, 2010, pp. 301–302.
9. E. F. Moore, “Gedanken-experiments on sequential ma- chines,” *Automata studies*, vol. 34, pp. 129–153, 1956.
10. G. H. Mealy, “A method for synthesizing sequential cir- cuits,” *Bell System Technical Journal*, vol. 34, no. 5, pp. 1045– 1079, 1955.