저작자표시-비영리-변경금지 2.0 대한민국

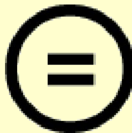
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

Master’s Thesis of Yong-Hwan Yoo

Snapshot-based Migration of ES6 JavaScript

ES6 자바스크립트의 스냅샷 기반 마이그레이션

August 2021

The Graduate School
Seoul National University
Department of Electrical and Computer Engineering

Yong-Hwan Yoo

Snapshot-based Migration of ES6 JavaScript

Soo-Mook Moon

Submitting a master's thesis of
Electrical and Computer Engineering

August 2021

The Graduate School
Seoul National University
Department of Electrical and Computer Engineering

Yong-Hwan Yoo

Confirming the master's thesis written by
Yong-Hwan Yoo
Month Year

Chair	Yunheung Paek
Vice Chair	Soo-Mook Moon
Examiner	Byoungyoung Lee

Abstract

With the growing popularity of the web platform and JavaScript, an interesting user experience called application (app) migration has been proposed for JavaScript programs. To enable a non-breaking workflow across different devices, recent studies have proposed snapshot-based techniques in which an app's runtime state is serialized into a text form that can be restored back later. A limitation of existing literature, however, is that they are based on old JavaScript specifications. Since major updates introduced by ECMASCript2015 (a.k.a. ES6), JavaScript supports various features that cannot be migrated correctly with existing methods. Some of these features are heavily used in today's real-world apps and thus greatly reduces the scope of previous works.

In this thesis, I will mainly introduce my work presented in [19]. In the paper, we analyzed ES6 features such as block scopes, modules, and class syntax that were previously uncovered in app migration. We presented an algorithm that enables migration of apps implemented with these new features. Based on the standards adopted in modern JavaScript engines, our approach serializes a running program into a scope tree and reorganizes it for snapshot code generation. We implemented our idea on the open source V8 engine and experiment with complex benchmark programs of modern JavaScript. Results showed that our approach correctly migrates 5 target programs between mobile devices. Our framework could migrate the most complex app of source code size 213KB in less than 200ms in a X86 laptop and 800ms in an embedded ARM board, showing feasibility in resource-constrained IoT devices. I will also discuss possible use cases and research directions and conclude.

Keyword : JavaScript, app migration, serialization, code generation
Student Number : 2019-26414

Table of Contents

Abstract i
Table of Contents ii
List of Tables iii
List of Figures iii

Chapter 1. Introduction 1
1.1. JavaScript App Migration 1
1.2. Purpose of Research 2

Chapter 2. Background 4
2.1. Snapshot-based Approach 4
2.2. Function Closure and Scope Tree 6
2.3. Limitations of Previous Works 6

Chapter 3. Proposed Approach 10
3.1. Module Profiling 10
3.2. Migrating Modified Built-in Objects 11
3.3. Scope Tree Building 11
3.4. Syntax-Aware Tree Re-ordering 12
3.5. Tree Partitioning 13
3.6. Snapshot Code Generation 13

Chapter 4. Evaluation 17
4.1. Implementation and Setup 17
4.2. Scope Tree Analysis 18
4.3. Snapshot Code Sizes 19
4.4. Framework Time Overhead 20

Chapter 5. Discussion 22
5.1. Limitations 22
5.2. Alternative Approach 22
5.3. Potential Use Cases 23

Chapter 6. Conclusion 24

Bibliography 25

Abstract in Korean 27

List of Tables

Table 1. Target Programs for App Migration........................ 18

Table 2. Scope Tree Details.. 19

Table 3. Code Size Across Migration 19

List of Figures

Figure 1. Scope Chain and Scope Tree Example 5

Figure 2. ES6 Module Example 7

Figure 3. Incorrect Restoration of Class Definitions............... 8

Figure 4. High−level Overview 10

Figure 5. Tree Partitioning Example (UniPoker) 13

Figure 6. Pseudo code for Code Generation.......................... 14

Figure 7. Scope Tree Example (ML benchmark) 20

Figure 8. Breakdown of Our Framework's Overhead (ms) ... 21

Chapter 1. Introduction

1.1. JavaScript App Migration

Among various modern programming languages, JavaScript remains as one of the most pervasive scripting languages. According to recent survey results from StackOverflow [①], its popularity remains unsurpassed for several consecutive years. An important factor for this popularity is its compatibility with web browsers which are available in most mobile devices by default. Moreover, adoption of JavaScript outside web browsers has given rise to server-side or desktop apps that run in popular runtime environments such as Node.js or electron. Also, several smart device vendors support built-in web browers (e.g. Samsung Tizen, LG webOS), thus readily running web applications written in JavaScript.

Wide platform pool of JavaScript makes it suitable for a cross-device computing concept called liquid software [5, 6, 16] in which the workflow of interactions and services are continued across devices. While similar approaches were proposed for native mobile platforms [1], they lacked support for devices from different vendors. Yet, the high portability of web apps and freedom from predatory control of OS vendors exempt them from such issues.

With a similar concept, [3, 13, 14, 15] proposed app migration frameworks for stateful web apps, in which browser sessions can be migrated across devices. Their main approach is to profile a running program's states, such as the objects in JavaScript heap, and saving them into a text-formatted file (i.e. snapshot). Generating a snapshot as a JavaScript code enables a low-overhead framework for continuous user experience across a heterogeneous device pool. Later studies extended the techniques to IoT [8] and compute offload [7, 11], suggesting novel use cases like multi-device web games and collaborative machine learning in browsers.

[①] https://insights.stackoverflow.com/survey/2020/

In order to implement app migration for JavaScript apps, previous works addressed important challenges raised by the dynamic nature of JavaScript. Notably, [15] suggested solutions for saving variables hidden inside a function closure. [13] extended this and proposed a scope tree building algorithm to save complex scope hierarchy of function closures. However, JavaScript language, as well as its ecosystem, has evolved significantly and has continuously been refined on a yearly basis. Modern JavaScript engines support by default various features[②] used in real-world web apps (e.g. slack, ebay, duckduckgo). These apps make heavy use of new language features (e.g. block scoping, class, module and new built-in types) introduced in ECMAScript2015 standards [③]. This raise non-trivial issues to all prior works which, at their best, are based on the old ECMAScript5.1 standards.

1.2. Purpose of Research

Our work [19] tackles the problem of migrating runtime states of ES6 JavaScript programs. We analyze the major language features defined in ES6 specifications and discuss the main challenges in app migration. Our work expands scope tree building by [13] to support two new variable scopes introduced by block scoping and module system. Based on analyses of scope trees, we propose methods for restoring class syntax included in modern JavaScript programs together with new built-in types. We implement our work as a JavaScript module and tested our idea using 5 modern benchmark programs. Experimental results in two different mobile devices show that our approach correctly migrates all programs with minimal overhead, suggesting feasibility of in resource-constrained environments. In short, our paper following made contributions:

- To the best of our knowledge, it is the first study on runtime migration of ES6 JavaScript. We analyze the challenges raised by

---

② https://kangax.github.io/compat-table/es6/
③ http://www.ecma-international.org/ecma-262/6.0/

new languages features and propose new ways to serialize and restore their states.

- We evaluate our work on complex benchmark programs written in ES6. Experiments in 2 different mobile devices show the app migration causes low time overhead and is thus feasible in resource−constrained devices.
- We show that the size of restoration code generated by our framework is comparable to previous state−of−the−art. We further analyze different ES6 programs based on the generated scope trees and snapshot codes.

Chapter 2. Background

2.1. Snapshot−based Approach

Several recent works on JavaScript app migration have proposed capturing the application state at JavaScript level [3,14,15,13,8]. As JavaScript engines save variables of the global scope as the global object' s properties, their values can be accessed by enumerating these properties cleverly. After each element state is serialized at a source device, a snapshot code can be generated to restore their values at the target device. When this code is executed at the target device, original global scope state can be migrated with minimal overhead, allowing the user to resume execution of the app from the serialized state. Since this simplifies the process of restoring an app as opposed to native−level solutions [9, 2], we follow the state−of−the−art approach proposed by [13] and incrementally build on this baseline study throughout the paper.

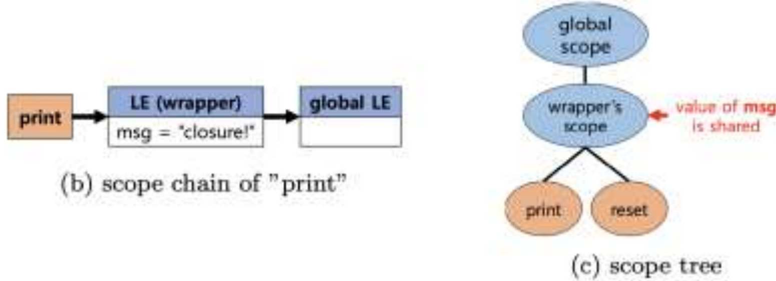
2.2. Function Closure and Scope Tree

During a program' s runtime, the JavaScript engine manages a call stack to save the context, a.k.a. the execution context, in which the code is executed. Each execution context consists of a lexical environment (LE) whose environment record saves the set of variables, functions, etc. bound to the LE. Another component of an LE is a reference to the outside LE, which is referred to as "outer", together defining the variable' s scope.

As a JavaScript code makes some function call, the JavaScript engine dynamically creates a new execution context and a corresponding LE. Then, the function is dynamically bundled with its outer LE in which it is defined as a closure. These closures are discussed as a major challenge in previous works on JavaScript app migration. In fact, to preserve a function' s state completely, we

```
1  function wrapper() {
2      var msg = 'closure!';
3      function print() { console.log(msg); }
4      function reset() { msg=''; }
5      return {print, reset};
6  }
7  var {print, reset} = wrapper();
```

(a) source code



(b) scope chain of "print"

(c) scope tree

Figure 1. Scope Chain and Scope Tree Example

need to save and restore the “outer” LE accessed by the function’s closure which is internally managed by JavaScript engines. To tackle this issue at JavaScript level, [15] modified the JavaScript engine to gain direct access to the internal property ’Scope’ to recursively obtain the chain of LEs, a.k.a. scope chain. This enables a function to be restored together with the original context.

As demonstrated in later works, however, migrating functions is more challenging when multiple closures share the same LEs. For example, Figure 1 illustrates two functions print and reset that reference the same variable msg via their closures. In this case, simply saving the LE of each function will not restore the whole program correctly. If the relationship between two closures is not captured, restoring each scope chain will generate multiple copies of shared contexts. Thus, the whole scope hierarchy needs to be serialized to prevent unexpected breakdowns in mysterious cases.

To mitigate this problem, [13] proposed combining all the scope chain information into a single data structure called scope tree. Figure 1c shows an example scope tree generated for Figure 1. In this scope tree, “print” and “reset” become child nodes of the same node because they are defined inside the local scope of

wrapper. Afterwards, traversing this tree in pre-order generates a restoration code for the original program state.

2.3. Limitations of Previous Works

Although previous works incrementally addressed important issues in JavaScript app migration, modern JavaScript apps rely on newer complex features to which existing approaches do not provide solutions. We analyzed new specifications in ES6 standards not addressed in previous approaches, specifically focusing on 4 features prevalent in modern JavaScript apps and frameworks: block scoping, module system, class syntax, and new data types.

2.4. ES6 Features and Issues in App Migration

Block Scoping Prior to ES6, a JavaScript variable was declared with keyword var and scoped to the innermost surrounding function (a.k.a. function-scoped). This means variables were available anywhere within the function it is declared. On the other hand, ES6 introduced a new variable type called block-scoped variables as a core update. Declared with keywords let or const, these variables follow a more common convention of other languages and are scoped to any innermost block that surround them and cannot be accessed until their lexical bindings are evaluated.

Module System ES6 standards introduced a new module system to JavaScript which allows splitting a large piece of code into multiple files using built-in syntax. This replaced previous platform-specific module implementations[4][5] and has been shipped into all major JavaScript engines as the de facto standard. A module's code is stored in a separate file, each containing a set of declarations and

---

[4] http://www.commonjs.org/
[5] https://github.com/amdjs/

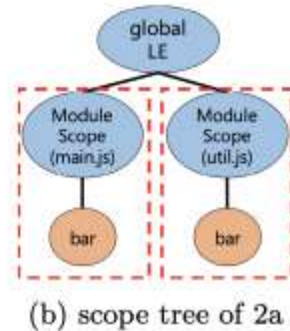
util.js

```
1  var foo = 'util-foo';
2  export function bar(){
3      print(foo);
4  }
```

main.js

```
1  import {bar} from './util.js
       ';
2  var foo = 'main-foo';
3  bar();  // 'util-foo'
```

(a) source code



(b) scope tree of 2a

Figure 2. ES6 Module Example

statements which may be accessed from another module as read-only. In practice, programmers specify an entry point module and explicitly load it, for example, via a ⟨script type="module"⟩ tag into an HTML file of a web app.

JavaScript engines execute modules differently from regular scripts, so the same code can have different semantics depending on whether it is loaded as a script or module. For example, calling this keyword at top-level will give different results in a script (global object) and module (undefined). Thus, ES6 app migration needs to preserve such semantics of modules.

Internally, the JavaScript engine creates a new LE, a.k.a. ModuleScope whenever a new module's code is executed, thereby isolating each module's scope. In perspective of a scope tree, this means that each module's top-level bindings are saved in a separate node. The challenge here is that the scope tree alone cannot capture the order of each module's declaration (i.e. relationship between the modules). For example, Figure 2b shows scope tree generated for the code of Figure 2a. Both ModuleScope nodes save the same function bar, but we cannot identify in which module this function was first declared.

```
1  var Circle;
2  var Shape;
3  Shape = class {
4      constructor(x,y){
5          this.x = x
6          this.y = y
7      }
8  }
9  Circle = class extends Shape {
10     constructor(x,y,r) {
11         super (x,y)
12         this.r  = r
13     }
14 }
15 var c = new Circle();
```

(a) source code

```
1  var Circle = class
      extends Shape {
2      ...
3  } // syntax error
4  var Shape = class {
      ... }
5
6  var c = new Circle();
```

(b) wrong snapshot code

Figure 3. Incorrect Restoration of Class Definitions

Class Syntax ES6 also defined class definitions as a special function type while reserving several keywords to mimic syntax like class-based languages on top of JavaScript's object-based nature. A class's constructor function is differentiated from normal functions and is bound to a new BlockScope generated for that class. Subclassing in ES6 classes is done with the extends keyword, for which the JavaScript engine evaluates the parent class and dynamically links the child class's constructor and prototype to their parent class counterparts.

In order to save and restore class definitions for app migration, naively treating them as regular functions and capturing their states using a scope tree will not preserve the syntactic order between different classes. More specifically, the extends keyword used in class subclassing requires that a parent class's BlockScope is evaluated before executing a child class's BlockScope.

Due to JavaScript's dynamic nature, however, such order between subclassing classes are often not captured automatically. Figure 3a is a source code of two class variables whose declarations (line 1-2) and definitions (line 4-5) are in different order. If we restore this app like using a scope tree [13], identifier Circle will be restored before Shape. Because dependencies

between the two identifiers are not explicit, a generated snapshot code (Figure 3b) will raise a syntax error when it is executed (line 1). Identifying such dependencies becomes more challenging if classes are defined in different scopes (e.g. in different lexical blocks) or if parent classes are defined by arbitrary expressions. Thus, we need a new strategy that can generalize and account for such new syntax.

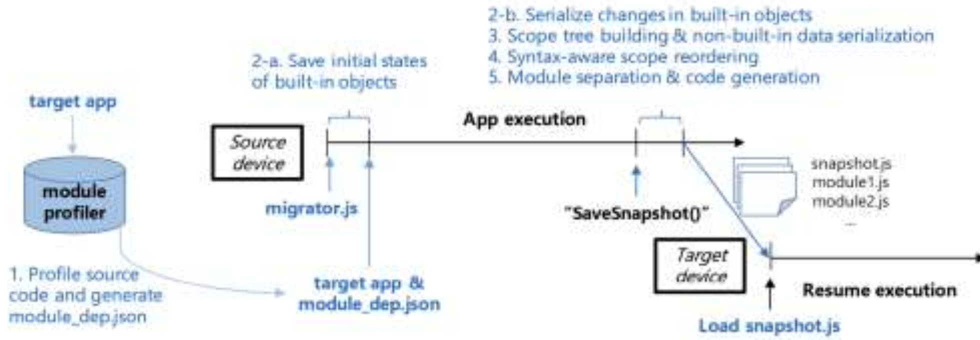


Figure 4. High-level Overview

Chapter 3. Proposed Approach

In Figure 4, we show a high-level overview of our framework presented in [19]. Given a target app, we save its module structure into a JSON file (“module dep.json”) with a lightweight static analysis (module profiler) which is loaded together for later use. The app user can trigger app migration by calling a global function (“SaveSnapshot”) of our framework (“migrator.js”) to generate a snapshot code (“snapshot.js”). The user can load this snapshot code at the target device and restore original app state in the source device so that app execution can continue seamlessly. We now explain details of each stage.

3.1. Module Profiling

To restore a module structure in the target device, our framework statically analyzes the app source code and saves dependencies between different modules included in the app. This is because a static analysis can capture any complex relationship between ES6 modules that cannot be captured easily at runtime, such as two different JavaScript modules that have cyclic dependencies. As such, we add a module profiler stage in advance to app deployment. Given an app’ s source code and its entry

module name, we generate a JSON file (“module dep.json”) that saves the dependency graph between modules. This dependency graph models each module as a node and variables imported to each module as an incoming edge. This file is later loaded with the target app to restore the relationship between different modules.

3.2. Migrating Modified Built-in Objects

Before loading some target app, our framework saves the initial states of JavaScript’ s built−in objects such as Array, String, etc. This is an optimization to efficiently migrate JavaScript built−in objects based on the intuition that after a JavaScript engine initializes their properties, most built−in objects are rarely modified during program execution [12, 11]. Inspired by this observation, we do not serialize the unmodified properties redundantly and instead restore them at the target device via default engine startup. To save the other modified portion and minimize our snapshot code size, our framework loads our app migration script before the actual app is loaded by the JavaScript engine and immediately save initial states of
built−in objects (step 2−a). At the actual app migration, we traverse these built−in object once again to find the properties that are modified from their initial states during app execution (step 2−b) and generate a JavaScript code that restores these changes via assignment.

3.3. Scope Tree Building

Our framework saves global identifiers (e.g. variables, objects) that were created during app execution, together with their values and properties. If some object is found to be a function (LeafFunction), we traverse the scope chain and collect scope information (i.e. LEs) recursively up to the outermost scope (GlobalScope). At the end of each traversal, we update the scope tree with the collected information as in previous approaches [13]

so that closure variables and their relationships are serialized. At the end of this stage, the resulting scope tree can be composed of 5 node types which we abbreviate as following:

- G = GlobalScope; global scope of a program.
- M = ModuleScope; top-level scope of a module.
- F = FunctionScope; scope introduced by a function.
- B = BlockScope; scope introduced by a block statement.
- L = LeafFunction; function that starts a scope chain.

3.4. Syntax-Aware Tree Re-ordering

Once a scope tree is generated, we collect dependency information between tree nodes to address the issue raised by subclassing syntax in class definitions. More specifically, this dependency defines the order in which a parent class and a child class will be declared so that the extends clause (i.e. reference to the parent) in every class definition is evaluated without syntax error. By re-ordering the scope tree with respect to such dependency, we ensure that a parent class is present with the right values when evaluating the child.

Once again, the challenge here is finding relationship between these class constructors, i.e. finding each class' s parent. Here, we exploit the prototype-based inheritance model of ES6 classes and inspect the internal links between JavaScript classes to find each class' s parent. Since every object in JavaScript, including class definitions, has an internal property named 'Prototype', recursively following these links up to null give us its "prototype chain". Based on this principle, we first iterate all observable class constructors and their parent class and locate their least common ancestor node in our scope tree. We then rearrange the two child branches to which the classes are bound, so that a pre-order depth first search reaches the parent class' s scope before the child' s.

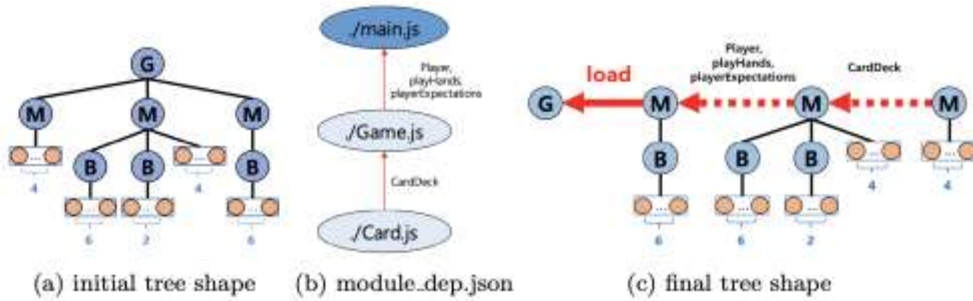


Figure 5. Tree Partitioning Example (UniPoker)

3.5. Tree Partitioning

As the next step, we restore the original partitioning of the source code so that each module's code can be generated separately. Since every module creates its own LE whose "outer" element points at the global LE, a ModuleScope node in our scope tree forms its own subtree as a direct child of GlobalScope node. Separating each module is straightforward: we iterate children of the root node to find all ModuleScope nodes and split their subtrees from the original scope tree.

To restore the relationship between module partitions, we examine the declarations saved in each module partition's root node (i.e. ModuleScope node) and recursively map each ModuleScope to a node in our previously saved dependency graph ("module dep.json"), thus restoring the original relationship between them. Finally, generating a glue code (e.g. ⟨script type="module"⟩) to load the entry module into global scope restores the original module structure of the application. As an example, Fig 5 shows tree partitions of a target app called UniPoker, originally composed of three code fragments.

3.6. Snapshot Code Generation

After our scope tree is reordered and partitioned, we generate a snapshot code for each scope tree partition. The final result of this stage will be multiple JavaScript files, each corresponding to a tree

```
Algorithm 1: Code Generator
1  function generateScope(node):
2  |  switch node.type do
3  |  |  case BlockScope do
4  |  |  |  code += "{"
5  |  |  |  foreach variable ∈ node.LE do
6  |  |  |  |  code += "let" + serialize(variable)
7  |  |  |  end
8  |  |  |  foreach childNode ∈ node.children do
9  |  |  |  |  generateScope (childNode)
10 |  |  |  end
11 |  |  |  code += "}";
12 |  |  end
13 |  |  case FunctionScope do
14 |  |  |  code += "(function(){"
15 |  |  |  foreach variable ∈ node.LE do
16 |  |  |  |  code += "var" + serialize(variable)
17 |  |  |  end
18 |  |  |  foreach childNode ∈ node.children do
19 |  |  |  |  generateScope (childNode)
20 |  |  |  end
21 |  |  |  code += "})"
22 |  |  end
23 |  |  case LeafFunction do
24 |  |  |  if node.function.startsWith("class") then
25 |  |  |  |  code += serializeClass(node.function)
26 |  |  |  else if node.function.prototype == undefined then
27 |  |  |  |  code += serializeMethod(node.function)
28 |  |  |  end
29 |  |  |  else
30 |  |  |  |  code += serializeFunction(node.function)
31 |  |  |  end
32 |  |  end
33 |  end
```

Figure 6. Pseudo Code for Code Generation

partition. Code for a partition can be generated by traversing the partition's nodes in pre-order and applying an appropriate code generation scheme to each visited node. We implement function generateScope which generates code for a given node and recursively invokes itself until a leaf node is reached. In other words, invoking generateScope with some partition's root node as argument returns its snapshot code. We show pseudo code of the code generator in Figure 6.

In line 2, we first check the type of the visited node to select an appropriate code generation scheme. If the node type is BlockScope

(line 3) or a FunctionScope (line 13), we generate a wrapper code corresponding to the scope type, which generates code for a given node and recursively invokes itself until a leaf node is reached. In other words, invoking generateScope with some partition' s root node as argument returns its snapshot code. We show pseudo code of the code generator in Figure 6.

In line 2, we first check the type of the visited node to select an appropriate code generation scheme. If the node type is BlockScope (line 3) or a FunctionScope (line 13), we generate a wrapper code corresponding to the scope type, i.e. lexical block statement (line 4 & 11) or an immediately invoked function expression (line 14 & 21). Inside the wrapper code, we serialize the value of each closure variable and invoke generateScope for each child node in the scope tree. Note that variables are declared differently in each case so that they are bound to the correct LE type.

When we visit a leaf node during a pre-order traversal (line 23), our code generator first checks if the function is a class (line 24), a method (line 26) or neither of two types (line 29). A class constructor is distinguishable lexically at JavaScript level by its keyword "class" while a method function is unique in that it does not have a "prototype" property. We again serialize its scope chain (i.e. node.function), process it with respect to the syntax of the type, and concatenate the resulting code.

In generating the snapshot code, our framework adds supports for new data types introduced in ES6 which cannot be serialized with existing methods. One example is the new primitive type **symbol,** whose value is created by calling the function Symbol() with an optional stringargument (i.e.key). Since each symbol saves a unique valueeven if generated with the same key, we cannot serialize thesedata types into strings directly like other primitive types (i.e.with JSON.stringify()). We therefore introduce an auxiliary array named sym_ref that saves all distinct symbol values found during the whole serialization process, and save the mapping from each symbol variable to one of these values. We later restore this sym_ref and generate a reference codes when the symbol value is

called.

Moreover, we propose an efficient way to migrate new standard built−in objects such as keyed collections (e.g. Map, Set). Whenever an occurrence of these types is found, our code generator adds a declaration an empty prototype of the built−in object and copies each element of the target object in the original insertion order using the corresponding built−in methods, e.g. Map.set("key", "value"). We apply a similar approach for a typed array object, but additionally save the subtype information (e.g. Int8Array). The target object can then be declared in the restoration code with this subtype information and restored in a similar fashion.

Chapter 4. Evaluation

4.1. Implementation and Setup

We used the V8 JavaScript engine of the open-source chromium browser to implement and evaluate our work, as it is currently the most popular platform adopted by major browser (Google Chrome, Microsoft Edge) and non-browser platforms (Node.js, electron). We cloned the source code of a recent version chromium browser (version 82.0.4060.0, Feb 15, 2020) to add accesses to internal 'Scope' property of functions. Our module profiler extends an open source npm 67 package built on Esprima JavaScript parser . The rest of the framework is implemented as a module named "migrator.js" so as to be easily plugged into other JavaScript apps for app migration support. "SaveSnapshot" function in the module is attached to a button click event to provide interactive interface.

We compiled our modified V8 engine to experiment in two environments: (1) a X86 laptop with Intel Core i7-7700 3.6GHz CPU and 32GB memory (2) ODROID-XU4 embedded board with ARM Cortex-A15 Quad 2Ghz & CortexA7 Quad 1.3GHz CPUs, and 2GB of memory, to simulate resource-constrained scenarios. We then adapted several programs from JetStream2 benchmark (Table 1) that show various real-world usages of ES6 features. Original details of the benchmark can be found in [4]. Since we couldn't find any standard benchmark for testing ES6 modules, we additionally split source codes of two target programs (UniPoker and ML) into multiple modules.

We first downloaded source codes of the target apps, saved them in the source device, and adapted each app so that it imports our framework in advance of app loading. We loaded each app in our modified browser and executed app migration at 2 different execution points: (1) after a target app is fully loaded and (2) after program finished several iteration. The generated snapshot file is then loaded into a new browser session in a target device.

Table 1. target programs for app migration

app	description	ES6 features
UniPoker	5 card stud poker simulation using Unicode (U+1F0A1). 3 modules (Card.js, Game.js, main.js)	let/const, classes, new built-in types, module
Air	ES6 port of WebKit B3 JIT's allocateStack phase. Runs hot function bodies of popular benchmarks.	let/const, classes, new built-in types
Basic	ES6 implementation of ECMA-55 BASIC standard. Runs several simple apps (e.g. find prime numbers)	let/const, classes, new built-in types
Babylon	JavaScript parser used in Babel transpiler. Parses 4 JavaScript sources with intensive string processing.	let/const, classes, new built-in types
ML	Feedforward neural network for machine learning. Trains several networks with different activation functions and datasets. Refactored into 5 modules.	let/const, classes, subclassing built-in types, module

To ensure correctness of app migration, we first checked the runtime behavior of each benchmark program by resuming their execution after app migration multiple times. We also inspected all global identifiers in the original and new session and made sure their all values are preserved. Lastly for the 2 benchmarks written using ES6 modules, we checked if each module is properly split from each other with the correct import/export statements. Our inspection results showed that our snapshot codes restored the original program correctly.

4.2. Scope Tree Analysis

We summarized scope tree results of our benchmark programs in Table 2. Results of UniPoker and ML show that tree partitioning will yield 3 and 5 additional module partitions respectively, same as the original source codes. Compared to other programs, scope trees of Babylon and ML had relatively more complex structures (Fig. 7). For example, in the center of ML' s result we can observe a branch of length 5 (G-M-B-F-B) shared by 180 different leaf function nodes. We observed that most of the LE nodes in these complex tree structures are BlockScope nodes. In fact, a large portion of these nodes are created by class definitions and thus their child nodes are subject to syntax-aware re-ordering.

Table 2. Scope Tree Details

app	tree height	# of LEs				# of function
		G	M	F	B	
UniPoker	3	1	3	0	3	31
Air	2	1	0	0	14	251
Basic	2	1	0	0	7	50
Babylon	3	1	0	1	30	290
ML	5	1	5	1	43	598

Table 3. Code Size Across Migration

app	source (KB)	snapshot1 (KB)	snapshot2 (KB)
UniPoker	16	24	26
Air	403	625	626
Basic	45	68	68
Babylon	238	514	622
ML	213	644	644

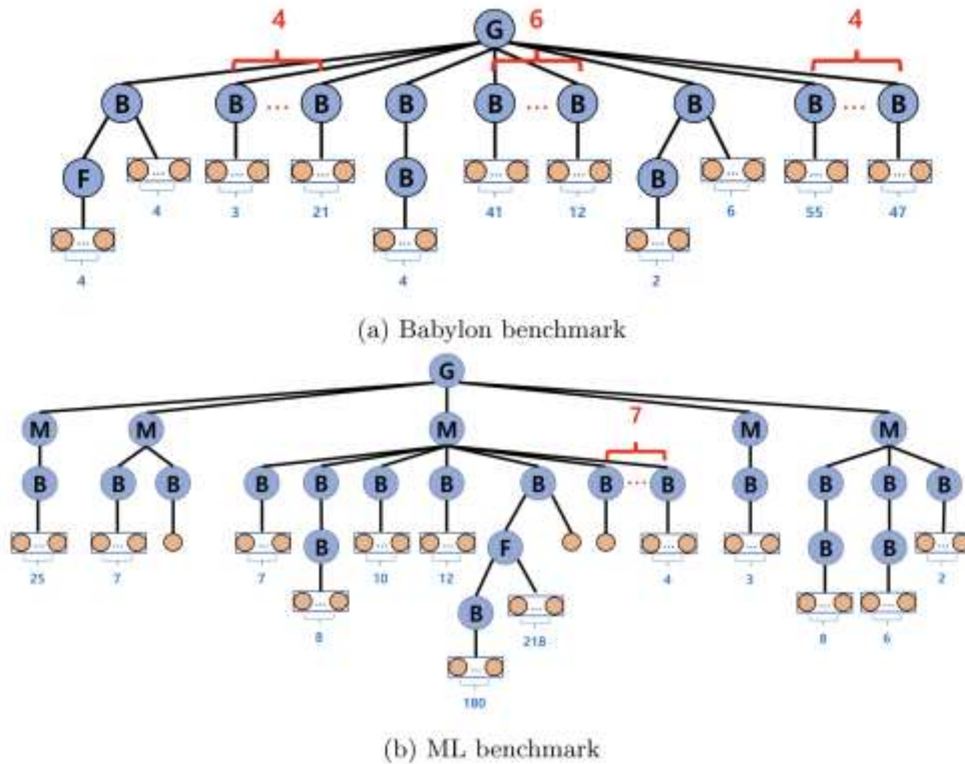
One of our framework's limitations lies in supporting asynchronous features in JavaScript (e.g. Promise API) which is outside the scope of this paper. For now, we simply disable app migration when such features are detected and leave support for them as future research direction. Yet, it is worth mentioning that these features essentially do not add new scoping rules and thus will not break the semantics of our overall scope serialization process.

4.3. Snapshot Code Sizes

In our framework, a small snapshot code size is desirable because it can reduce time to transmit the snapshot code between devices and shorten app loading time in the target device. Table 3 shows the source code and snapshot code size at the two execution points. Intuitively, size increase will be relatively larger for complex scope trees with more LEs, since we restore each LE by generating reference codes that is not present in the original source.

Unlike other benchmarks whose snapshot code sizes are mostly consistent across all execution points, noticeable increase exists between snapshot1 and snapshot2 in Babylon. This is because in between the two snapshot points, Babylon read 4 JavaScript source codes from external files and loads them into memory, thus greatly increasing program state size.

Another noticeable observation is that code size increase is unusually larger in ML compared to any other program. This is because ML's source code heavily uses the eval() function for code compression, which has long been deprecated. Even including



(a) Babylon benchmark

(b) ML benchmark

Figure 7. Scope Tree Example (ML benchmark)

such exceptional cases, the snapshot codes sizes are only 2.01X larger than the source code on average. This is comparable to previous state-of-the-art baseline result by [13], which reports 1.97X code size increase for the Octane benchmark 2.0 based on ES5 syntax. Considering the extra lines of code added to support ES6 features (e.g. glue codes for restoring dependencies in modules and classes), we conclude that the snapshot size is reasonably small even for resource-constrained devices.

4.4. Framework Time Overhead

Loading the framework and serializing initial built-in object states was consistent across benchmarks: 93ms (std 1ms) in laptop and 346ms (std 2ms) in ARM board, i.e. initial steps take similar times regardless of target apps. Fig 8 shows additional time overhead imposed by each stage. Total time spent for snapshot generation is

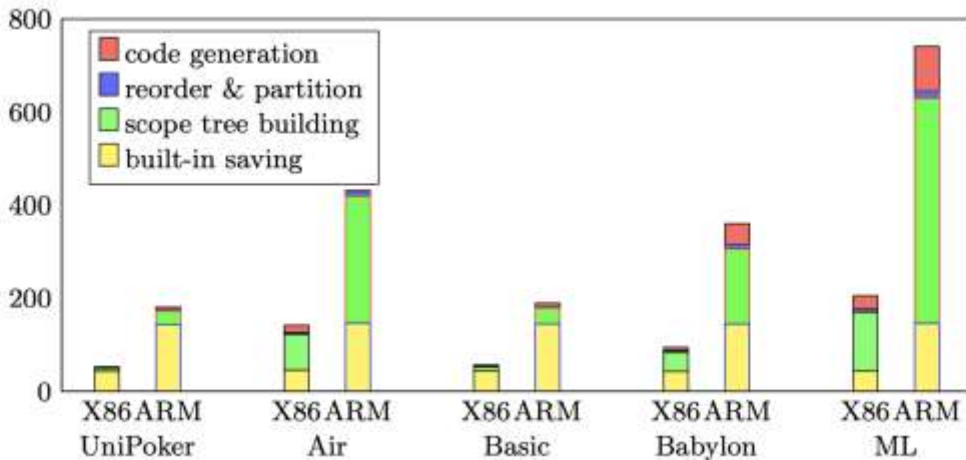


Figure 8. Breakdown of Our Framework's Overhead (ms)

less than 200ms in laptop and 800ms in ARM board in the worst case (ML). This is considered small enough for continuous progression in multi-device experience [13] and even for single-device experience [10]. Thus, the time overhead is small enough to provide seamless experience across devices from a user-centric perspective, and feasible in resource-constrained environments.

While built-in saving time is measured almost the same throughout all programs, the other 3 measurements are dependent on the source program itself and thus largely different by program. This is consistent to that of code sizes (i.e. the larger the snapshot size, the longer the framework took to generate it). Among the three items, time spent in reorder & partition was substantially lower than other stages in all cases, implying that extra steps for migrating ES6 features causes minimal extra overhead.

Chapter 5. Discussion

5.1. Limitations

This paper mainly discusses methods to save and restore major ES6 features such as block scoping, class definition, module structure, and several new built-in types. In addition to those discussed in this paper, ES6 standards introduced several popular features including arrow functions, new operators (rest, spread, for-of), template literals, and destructuring assignments. Since these features do not cause complex issues in app migration at function-level granularity, they are trivially supported by our framework.

Our framework does not fully support migrating unresolved states of asynchronous language features like promises and async/await patterns. These two feature states are managed by engine internals that cannot be serialized easily at JavaScript level. The current implementation simply limits app migration when an un-resolved promise is present, but we leave this as an important future research direction. For other features in more recent ECMAScript standards that is not covered in this paper (e.g. BigInt type), we expect them to be supported easily since those minor updates do not break the semantics of our approach. For now, we leave it as future work.

5.2. Alternative Approach

Our work modifies the JavaScript engine source code in order to fully serialize a function closure. This design choice only requires minimal modifications to the engine source code (63 extra lines of code). Moreover, it is a common practice for smart device vendors to use their own customized version of major JavaScript engines, e.g. Samsung’s Tizen platform is implemented on the V8 engine.

An orthogonal approach proposed by [8] and [14] instruments

the target app’s source code (a.k.a. instrumentation-based approach) so that the scope chain is exposed explicitly. An important issue of these approaches, however, is that the instrumentation process increases the original code size drastically. For example, results from a recent work [8] show that the source code size may increase by 74X in extreme cases. This greatly slows down the app loading and execution in resource-constrained devices. In contrast, our approach provides a more practical solution for migrating large apps between IoT devices with minimal overhead.

5.3. Potential Use Cases

This paper applies a snapshot-based approach to enable an interesting user experience called app migration. However, techniques discussed in our work can give rise to numerous other opportunities for JavaScript-based apps. One such example is [17] by Yeo et al. which applied a snapshot-based technique to shorten loading time of webapps. Their experiments show that saving a snapshot of a web app and using it as the app loading point can save 77% of the initialization time. Their work was extended further by in [18] to cover challenging apps such as those with non-deterministic behavior. Combined with these works, our work can extend the idea to reducing loading time of modern web applications.

There are also other studies that applied snapshot-based approaches for designing distributed systems. Gascon-Samson et al. [8] designed a system in which stateful JavaScript app states can be transferred between different IoT devices in the form of a snapshot. Jeong et al. [11] proposed a snapshot-based approach towards offloading intensive web app computations from resource-constrained devices to cloud servers with minimal overhead. They implement the idea to seamlessly offload compute-intensive webapp workloads using HTML5 web workers. Our work can extend these works with minimum overhead for supporting ES6 features.

Chapter 6. Conclusion

In my work on snapshot-based migration of ES6 JavaScript, we addressed challenges in snapshot-based app migration of ES6 JavaScript programs. We analyzed various features in ES6 standards and proposed methods to handle them efficiently, including manipulation of the scope tree and code generation for new scope types and data types. We implemented our proposal on the open-source V8 engine as an easily pluggable module. Evaluation on complex ES6-based benchmark programs shows that our framework can generate reasonable size snapshots with little time overhead in mobile devices, which shows feasibility in resource-constrained IoT devices. By combining static and dynamic methods, our framework achieves better efficiency than purely static methods and can potentially be extended to web loading time acceleration or IoT applications.

Bibliography

[1] Apple: Handoff for developers (2018), https://developer.apple.com/handoff/

[2] Barr, E.T., Marron, M., Maurer, E., Moseley, D., Seth, G.: Time-travel debugging for javascript/node. js. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 1003–1007 (2016)

[3] Bellucci, F., Ghiani, G., Paterno , F., Santoro, C.: Engineering javascript state persistence of web applications migrating across multiple devices. In: Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems. pp. 105–110 (2011)

[4] browserbench:Jetstream2benchmark(2019),https://browserbench.org/JetStream/

[5] Gallidabino, A.: Liquid web architectures. In: International Conference on Web Engineering. pp. 560–565. Springer (2019)

[6] Gallidabino, A., Pautasso, C.: The liquid. js framework for migrating and cloning stateful web components across multiple devices. In: Proceedings of the 25th International Conference Companion on World Wide Web. pp. 183–186 (2016)

[7] Gallidabino, A., Pautasso, C.: The liquid web worker api for horizontal offloading of stateless computations. Journal of Web Engineering pp. 405–448 (2019)

[8] Gascon-Samson, J., Jung, K., Goyal, S., Rezaiean-Asel, A., Pattabiraman, K.: Thingsmigrate: Platform-independent migration of stateful javascript iot applications. In: 32nd European Conference on Object-Oriented Programming (ECOOP 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018)

[9] Google:Customstartupsnapshots,https://v8.dev/blog/custom-startup-snapshots

[10] Google: Measure performance with the rail model (2020), http://web.dev/rail

[11] Jeong, H.J., Shin, C.H., Shin, K.Y., Lee, H.J., Moon, S.M.: Seamless offloading of web app computations from mobile device to edge clouds via html5 web worker migration. In: Proceedings of the ACM Symposium on Cloud Computing. pp. 38–49 (2019)

[12] Kwon, J.w., Lee, H.J., Moon, S.M.: Webdelta: Lightweight migration of web applications with modified execution state. In:

International Conference on Web Engineering. pp. 435–450. Springer (2020)

[13] Kwon, J.w., Moon, S.M.: Web application migration with closure reconstruction. In: Proceedings of the 26th International Conference on World Wide Web. pp. 133–142 (2017)

[14] Lo, J.T.K., Wohlstadter, E., Mesbah, A.: Imagen: Runtime migration of browser sessions for javascript web applications. In: Proceedings of the 22nd international conference on World Wide Web. pp. 815–826 (2013)

[15] Oh, J., Kwon, J.w., Park, H., Moon, S.M.: Migration of web applications with seamless execution. In: Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. pp. 173–185 (2015)

[16] Taivalsaari, A., Mikkonen, T., Systa ̈, K.: Liquid software manifesto: The era of multiple device ownership and its implications for software architecture. In: 2014 IEEE 38th Annual Computer Software and Applications Conference. pp. 338–343. IEEE (2014)

[17] JiHwan Yeo, JinSeok Oh, and Soo-Mook Moon. 2019a. Accelerating Web Application Loading with Snapshot of Event and DOM Handling. In Proceedings of the 28th International Conference on Compiler Construction (Washington, DC, USA) (CC 2019). Association for Computing Machinery, New York, NY, USA, 111-121.

[18] Jihwan Yeo, Changhyun Shin, and Soo-Mook Moon. 2019b. Snapshot-based Loading Acceleration of Web Apps with Nondeterministic JavaScript Execution. In The World Wide Web Conference. 2215–2224.

[19] Yoo YH., Moon SM. (2021) Snapshot-Based Migration of ES6 JavaScript. In: Brambilla M., Chbeir R., Frasincar F., Manolescu I. (eds) Web Engineering. ICWE 2021. Lecture Notes in Computer Science, vol 12706. Springer, Cham. https://doi.org/10.1007/978-3-030-74296-6_31

Abstract

최근 웹 플랫폼 및 자바스크립트의 인기와 함께, 자바스크립트로 작성된 프로그램을 위한 앱 마이그레이션 기술이 연구된 바 있다. 이는 이종의 기기 간에 연속적인 워크플로우를 제공해 새로운 사용자 경험을 제공하는 기술을 일컫는다. 여러 선행 연구에서 스냅샷 기반 방법론을 사용해 앱의 런타임 상태를 텍스트 형태로 직렬화 및 복원하는 시도를 했다. 그러나, 기존 연구들은 구 버전 자바스크립트 상에서 진행됐다는 한계가 있다. 이에 비해 ECMAScript2015 (ES6) 업데이트 이후 자바스크립트에 다양한 기능이 도입되었기 때문에, 기존 방법들은 오늘날 real-world 애플리케이션을 마이그레이션하기 어렵다.

본 논문은 [19]에서 소개된 우리의 프레임워크를 소개한다. 우리는 선행 연구에서 다뤄지지 않은 block scope, module, class syntax와 같은 ES6의 주요 기능을 분석했으며 이러한 새로운 기능을 사용하는 앱을 마이그레이션 하기 위한 알고리즘을 제안했다. 또한, 우리는 최신 자바스크립트 엔진에 대한 분석을 통해 실행 중인 자바스크립트 프로그램의 런타임 상태를 scope tree라는 자료구조 상에 직렬화하고, 후처리를 거친 scope tree로부터 스냅샷 코드를 생성했다. 이러한 방법론을 V8 자바스크립트 엔진인 상에 구현했으며, 복잡한 최신 자바스크립트 기능을 사용하는 벤치마크 프로그램에 대해 실험했다. 실험 결과를 통해 이러한 방법이 모바일 기기 간에 5개의 벤치마크 프로그램을 성공적으로 마이그레이션 시킨다는 것을 보였다. 복잡도가 가장 높은 앱 (ML 벤치마크, 소스 코드 크기 213KB)에 대한 실험에서 프레임워크로 인한 시간 부하를 측정한 결과, X86 랩톱에서 200ms 미만, ARM 기반 임베디드 보드에서 800ms 미만이었다. 이러한 결과를 통해 자원이 제한된 IoT 기기 등에 대한 적용 가능성을 검증했으며, 추가적으로 프레임워크의 활용 방안 및 향후 연구 방향에 대해 논의한다.

Keyword : 자바스크립트, 앱 마이그레이션, 직렬화, 코드 생성

Student Number : 2019-26414