Yapures: A Convenient Framework for Writing Safe JavaScript Code

Xiao Liu[O1], Yeoneo Kim[2], Sugwoo Byun[3], Gyun Woo[4]
[124]Dept. of Electrical and Computer Engineering, Pusan National University, Busan, Korea
[3]Department of Computer Engineering, Kyungsung University, Busan, Korea
[4]Smart Control Center of LG Electronics
e-mail:[124]{liuxiao, yeoneo, woogyun}@pusan.ac.kr, [3]swbyun@ks.ac.kr

Abstract

This paper introduces a framework named Yapures (Yet another PureScript) that provides a more convenient way to use PureScript for generating JavaScript code with type safety. The type safety and correctness of JavaScript are becoming more important since it can be run on the server side. There are several Haskell based systems to help developers to generate JavaScript code with type safety but most of them are difficult and cumbersome to use. Yapures is developed based on one of those systems called PureScript. By using Yapures, developers can write original JavaScript code and PureScript code together to generate type safety guaranteed JavaScript code.

1. Introduction

The uses of JavaScript are becoming more widespread in past a few years [1]. JavaScript was used only at the front-end of Web application for handling value transmission in DOM elements until a few years ago [2]. Since the Node.js was published, JavaScript and its plenty of server-side frameworks become to be used at the back-end [3].

Unlike at the front-end, the execution of programs at the back-end have to be absolutely correct which can make sure that the system runs well. Type safety is one of the most important practical foundations for programming language [4]. JavaScript was developed as a loosely-typed programming language at the first place because the type caused fault tolerances at the frontend of web are accessible to a certain extent. However, it will cause critical problems when the type errors occurred at the back-end.

There are several systems can generate JavaScript code with type safety such as GHCJS, Fay, and PureScript. All these systems are developed based on a functional programming language named Haskell. Since the feature of the outstanding type safety is a striking advantage obtained for free owing to Haskell, the generated JavaScript code through those systems supports type safety naturally.

However, using these systems to generate JavaScript code is cumbersome. Developers have to write specific kind source code in corresponding source code files, say .hs files for GHCJS and Fay, .purs files for PureScript.

This kind of mechanism constrains developers have to only write Haskell-like code to generate all JavaScript code they want, in somehow it could be more difficult on the contrary. Imagine that if a developer only want to generate a specific part of JavaScript code with type safety and combine it with other original JavaScript code, the Haskell-like code has to be written in a specific source code file and use the systems we mentioned to generate corresponding JavaScript code, then copy the generated code to the JavaScript code file contains other original JavaScript code.

Yapures provides a convenient approach that allows developers to write PureScript code in the original JavaScript code file directly. There is a code parser will parse the written source code and generate corresponding JavaScript code at the appropriate position of code file automatically. This kind of convenience can help developers to reduce their development time cost and make sure that they can only pay attention on how to writing their code without worrying that compositing generated JavaScript code with original ones.

This paper is organized as follows. Section 2

introduces the advantages and disadvantages of several similar approaches such as GHCJS, Fay, and PureScript. Section 3 illustrates the architecture of Yapures and demonstrates an example of how to use Yapures to writing safe typed JavaScript code. Section 4 presents the future work and Section 5 concludes.

2. Related Work

2.1. GHCJS

GHCJS is a Haskell-to-JavaScript transpiler wich can lift user input into a safe representation [5]. Developers can write Haskell-like code to generate JavaScript code with type safety through GHCJS. GHCJS itself is also built based on Haskell and GHC APIs. It supports many modern Haskell features including type system, multi-threads, and package management through Cabal. GHCJS also has new JavaScript-specific features such as JavaScript FFI (Foreign Function Interface) extension and synchronous/asynchronous threads.

However, as we mentioned, GHCJS is difficult to use because the code has to be written in a file with only GHCJS code, and it is inconvenience to put the generated JavaScript code to other JavaScript code together. Besides, not only the size of GHCJS system is quite large, but also the size of generated source code is quite large.

2.2. Fay

Similar with GHCJS, Fay is another Haskell based system that can compile Haskell-like code to JavaScript code [6]. Fay has several practical properties such as lazy evaluation, purity, size compressed JavaScript code generation, and auto-transcode valued between JSON files.

But still, Fay does not allow developers to mix the Haskell-like code with original JavaScript code in a same code file. Developers have to migrate the generated JavaScript code with original JavaScript manually.

2.3. PureScript

PureScript is a Haskell based small strongly typed programming language with expressive types that compiles to JavaScript. Comparing with the previous two systems, PureScript performs more powerful features. Unlike GHCJS and Fay, PureScript can be installed not only by stack tool but also by npm. PureScript also has its own environment configuration tool named pulp which can easily initialize and build PureScript project.

However, the disadvantages of PureScript are similar with GHCJS and Fay. PureScript does not allow developers to write two kinds of code in a same code file.

3. Yapures

Yapures is Yet another PureScript. Yapures allows developers to directly write PureScript code into the .js code file which contains other original JavaScript code. This feature makes sure that developers can only write the specific partial PureScript code they want, also, Yapures will automatically compile the corresponding part code and generate appropriate JavaScript code at the correct position to instead the PureScript code. Developers will no longer have to combine two parts of codes (generated one and original one) together manually anymore.

Yapures is developed with a modular architecture which is quite easy to optimize and maintain. Figure 1. illustrates all of the modules of Yapures. The basis of Yapures is a Node server which is for executing the script programs in Yapures. The code parser parses the mixed source codes including the original JavaScript code and PureScript code. The code generator and the lib generator take the parsed PureScript code and generate appropriate safe typed JavaScript code and libraries respectively. The code editor will remove the old PureScript code and insert new generated JavaScript code into the source code file at correct position.

Figure 1 demonstrates the architecture of Yapures. A Node.js server provides a execution environment for our system. The code parser parses the original JavaScript and PureScript mixed source code to extract the later one and send it to the code editor. Code editor reads the PureScript code and uses the lib generator and code generator to generates corresponding JavaScript code.

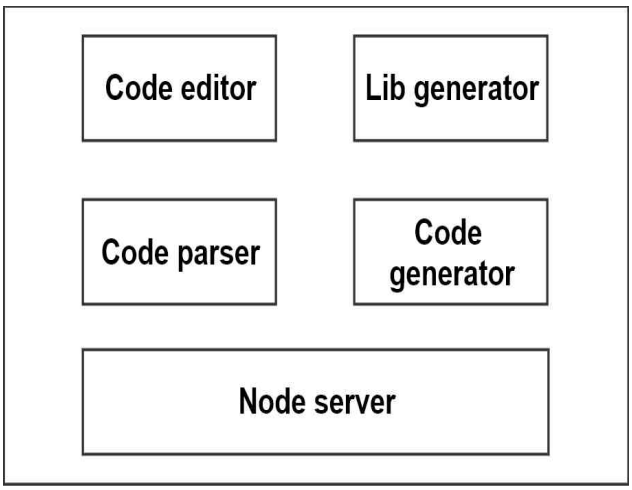

Figure 1 The modular architecture of Yapures

```
1  var saySth = function(){
2      console.log("something...")
3  }
4
5  //##
6  import Prelude
7  import Control.Monad.Eff.Console(log)
8  main = log "hello"
9  //##
```

Figure 2 PureScript code and original JavaScript code presented in Yapures

```
1  var saySth = function(){
2      console.log("something...")
3  }
4
5  // Yapures generated code
6  //##
7  "use strict";
8  var Prelude = require("../Prelude");
9  var Control_Monad_Eff_Console =
10 require("../Control.Monad.Eff.Console");
11 var main = Control_Monad_Eff_Console.log("hello");
12 main()
13 //##
```

Figure 3 The new generated code replaced the PureScript code

Figure 2 demonstrates how does Yapures present original JavaScript code and PureScript code. The codes from line 6 to line 8 are surrounded by two commented hashtags which can tell code editor module of Yapures the position of the PureScript code. After the executions of code parsing and new code generation, Figure 3 illustrates the generated safe typed JavaScript code at the appropriate position. The codes from line 8 to line 12 are the JavaScirpt code generated by Yapures with type safety.

4. Future work

We are planning to keep developing Yapures to make it more useful and convenient. We are going to add functions such as Web interfaces that the user can use Yapures through web browser without installing any programs.

5. Conclusion

This paper introduces a new framework that can use PureScript to generate JavaScript code with type safety in a convenient way. Yapures ensures that developers can write original JavaScript code and PureScript code in a same code file which can reduce the development time.

Acknowledgement

이 논문은 2018년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (2014-0-00035,매니코어 기반 초고성능 스케일러블 OS 기초연구 (차세대 OS 기초연구센터)).
*교신 저자 : 우균(부산대학교, woogyun@pusan.ac.kr).

Reference

[1] Flanagan, David. JavaScript: The definitive guide: Activate your web pages. “ O’Reilly Media, Inc.“, 2011.

[2] Goodman, Danny. Dynamic HTML: The Definitive Reference: A Comprehensive Resource for HTML, CSS, DOM & JavaScript. “ O’Reilly Media, Inc.“, 2002.

[3] Tilkov, Stefan, and Steve Vinoski. “Node. js: Using JavaScript to build high-performance network programs.“ IEEE Internet Computing 14.6: 80-83, 2010.

[4] Harper, Robert. Practical foundations for programming languages. Cambridge University Press, 2016.

[5] Mazumder, Mark, and Timothy Braje. “Safe Client/Server Web Development with Haskell.“ Cybersecurity Development (SecDev), IEEE. 150-150, 2016.

[6] Dijkstra, Atze, et al. “Building javascript applications with Haskell.“ SIAFL. Springer, 37-52, 2012.