International Journal of Advanced Science and Technology
Vol.119 (2018), pp.123-134
http://dx.doi.org/10.14257/ijast.2018.119.11

A Study on the Code Generator for a Virtual Machine Code based JavaScript Compiler

Jaehyun Kim and Yangsun Lee*

Dept. of Computer Engineering, Seokyeong University
16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 136-704, Korea
**yslee@skuniv.ac.kr*

Abstract

The JavaScript is a scripting language for web browser environments. It is used in various environments and fields besides web pages. To support the execution of these diverse JavaScript applications, we develop JavaScript compilers based on virtual machine code in the smart cross-platform. The smart cross platform is a platform that supports multiple programming languages and multiple platforms at the same time. It consists of a set of compilers and a virtual machine based on the intermediate language. The JavaScript compiler takes the JavaScript program as input and converts it into semantically equivalent intermediate code, SIL(Smart Intermediate Language) code. The virtual machine of a smart cross-platform takes the SIL code generated by the JavaScript compiler as input and outputs the result.

In this paper, we design and implement a code generator for the JavaScript compiler. The code generation process of the JavaScript compiler uses a code generator, a type analyzer, and a symbol table reflecting the information of the previous stage. First, the type analyzer returns the type of the input expression by referring to the type table, and is used in the semantic analyzer and the code generator. The code generator converts the input source code into a semantically equivalent intermediate language program based on the type analyzer, the symbol table, and the AST(Abstract Syntax Tree), and outputs the functions and prototype object accesses as a constructor. The virtual machine also supports basic dynamic types so that JavaScript, a dynamic type language, can be executed.

Keywords: *JavaScript Compiler, Virtual Machine Code, SIL Code, Smart Cross Platform, Smart Virtual Machine, Code Generator, AST(Abstract Syntax Tree), Visitor Pattern*

1. Introduction

The JavaScript [1,2] is a client-side scripting language that runs in a web browser. It is used for various purposes in various environments besides browsers. Originally, JavaScript was used to add dynamic functionality to web pages, but with the gradual development of the web environment, a variety of feature-rich web applications have emerged. Also, they are used in a variety of fields such as server programming, desktop applications, and mobile application programming [3-5].

To support the execution of these various JavaScript applications, we develop a JavaScript compiler based on virtual machine code for the smart cross platform. The smart cross platform [6-10] is a virtual machine system that supports multiple programming languages such as C, C++, Java, Objective-C and multiple platforms such as Windows, Linux, Android, iOS at the same time. It consists of a set of compilers and a virtual machine based on the intermediate language. We are designing and implementing a JavaScript

Received (April 6, 2018), Review Result (June 9, 2018), Accepted (June 20, 2018)
* Corresponding Author

compiler for smart cross-platform to enable rich JavaScript content to run on smart cross-platform platforms. The JavaScript compiler takes the JavaScript program as input and converts it into semantically equivalent intermediate code, SIL(Smart Intermediate Language).

In this paper, we design and implement a code generator for the JavaScript compiler [10-12]. The code generation process of the JavaScript compiler uses a code generator, a type analyzer, and a symbol table reflecting the information of the previous stage. The type analyzer parses and returns the data types of specific expression nodes, and is used in semantic analyzers and code generators. The code generator traverses the AST and outputs the SIL intermediate language, which is semantically equivalent to the input code, using a symbol table and a type analyzer. Since the JavaScript is a dynamic type language, the virtual machine must store the data type of each value at execution time. To accomplish this, we implemented basic dynamic type support in a smart cross platform virtual machine.

2. Related Studies

2.1. AST(Abstract Syntax Tree)

The AST is a tree that has the same meaning as the input source code and leaves only the necessary information, and is formed by removing unnecessary information from the parse tree in general. The AST contains important information of input source code and is smaller than parse tree, and so it is mainly used as intermediate representation in compiler [10,13-16].

The parse tree generated by the lexical analyzer and parser in the JavaScript compiler is converted to AST by the parse tree-AST converter. The AST continues to be used as an intermediate representation in place of the input source code in the symbol collector, code generator, and so on. The JavaScript compiler's AST is an irregular heterogeneous AST [12], which allows the structure to be understood only by the class definition, so the presentation method is consistent and easy to debug. In addition, this AST supports the use of the visitor pattern, so that the AST can be independently traversed and processed.

2.2. Visitor Design Pattern

The Visitor design pattern is a programming pattern that facilitates the addition of functionality to a data class that consists of a tree structure. It allows you to add new functionality that targets the same class without having to modify or recompile the data class by separating the functional class from the data class [17].

In the Visitor pattern, the function class traverses and processes the nodes of the data class and is implemented through the double dispatch of the Accept method of the data class and the Visit method of the Visitor class. Also, depending on the implementation, the Visit method can be configured to collect the processed information from each node of the data class [18].

The JavaScript compiler’s parser supports the Visitor pattern for the parse tree, which implements the parser-AST translator. In addition, AST designed in this paper also supports Visitor pattern, so new processing can be added without modification of intermediate representation.

2.3. SIL (Smart Intermediate Language)

The smart intermediate language (SIL) is an intermediate language for smart cross-platform [6-10]. SIL can accommodate both procedural programming languages and object-oriented programming languages, and can be an intermediate code for a variety of programming languages such as C, C ++, Java and Objective-C. It can also be run on smart virtual machines in a variety of platform environments that are not specific hardware specific. In addition, SIL names the type of command and the target data type in a consistent way, which is highly readable and easy to debug.

The SIL is a stack-based intermediate language that consists of basic instructions and optimization instructions. The basic instructions consist of instructions that perform basic operations, and can be divided into six categories as shown in Table 2. The optimization command associates one or more frequently occurring basic instructions with a single instruction, which improves the execution speed of the interpreter and reduces memory usage by the application. The JavaScript compiler compiles the input JavaScript program into SIL basic instructions, and the compiled SIL instructions is replaced by the optimizer with some instruction sequences as optimization commands.

Table 1. Classification of SIL Instructions

Operations category	Operations type	Count
Primary	Stack operations	59
	Arithmetic operations	87
	Flow control operations	20
	Type conversion operations	22
	Object operations	2
	Misc. operations	4
Optimization	Optimization operations	203

2.4. Smart Virtual Machine (SVM)

The Smart Virtual Machine (SVM) [6-10] is a smart cross-platform virtual machine that supports a variety of platforms including Windows, Linux, Android, iOS, and HTML5. The SVM is a stack-based virtual machine that can execute instructions written in SIL, the platform's intermediate language, and supports the execution of programs that use built-in libraries, thread scheduling, event handling, and so on. The smart virtual machine for each platform, such as Android and iOS, loads and executes the SIL code program compiled in the intermediate language.

Figure 1 shows the overall structure of the Smart Cross Platform and Smart Virtual Machine.

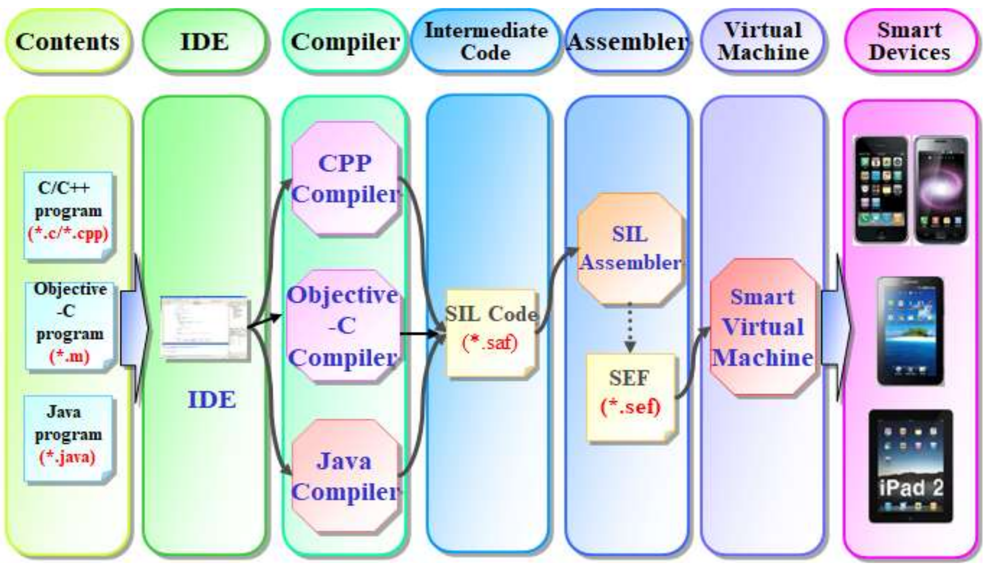


Figure 1. Overall Structure of the Smart Cross Platform

3. Code Generator for a JavaScript Compiler

3.1. Code Generator Model

The JavaScript compiler consists of a scanner, a parser, a parser-AST translator, a declaration processor, a semantic analyzer, a type estimator, and a code generator. Figure 2 shows the overall structure of the JavaScript compiler.

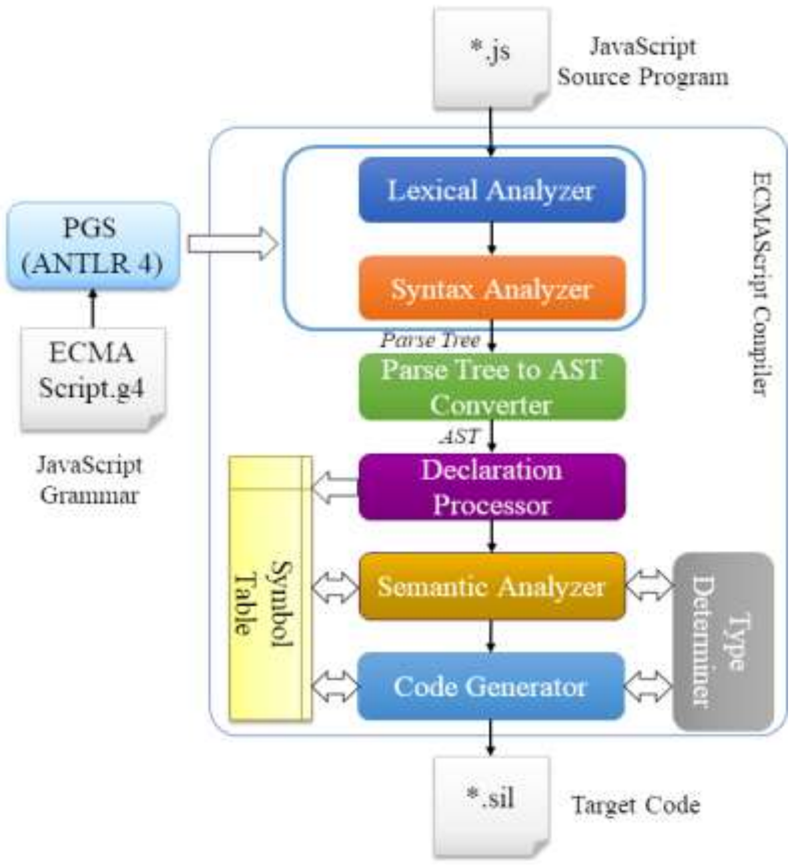


Figure 2. JavaScript Compiler Model

The JavaScript input program is converted to AST through a lexical analyzer, a parser, and a parse tree-AST converter. The generated AST is used as an intermediate representation within the compiler in the next step of the compiler. The lexical analyzer separates the input source code into a series of tokens, and the parser generates a parse tree by converting the tokens into a tree structure that conforms to the JavaScript syntax. The parse tree-AST converter removes unnecessary information from the parse tree and converts only the core information to AST. The designed AST supports the Visitor pattern and is defined to be able to work independently of the AST source code in the following process.

The declaration processor collects information about declared names of variables, constants, and functions to form a symbol table. The type analyzer analyzes the type of a given expression based on symbol information and AST, and is used in semantic analyzers and code generators. The semantic analyzer parses complex data types and semantics such as function return types, objects generated by constructors, and detects semantically incorrect parts to generate errors. The code generator takes the AST as an input and uses the symbol table and type analyzer to output the target code.

3.2. Code Generator

3.2.1. Type Analyzer

The type analyzer analyzes the type of expression and literal input based on AST and symbol information. The type analyzer is called by the semantic analyzer and the code generator. The JavaScript is a dynamic type language that sometimes knows the type information at compile time, but it can be known only at execution time. If the data type of the corresponding AST node can be more than one, "Var" type is returned. Otherwise, the type analyzer returns the type of the mathematical expression analyzed. Figure 3 shows the structure of the type analyzer.

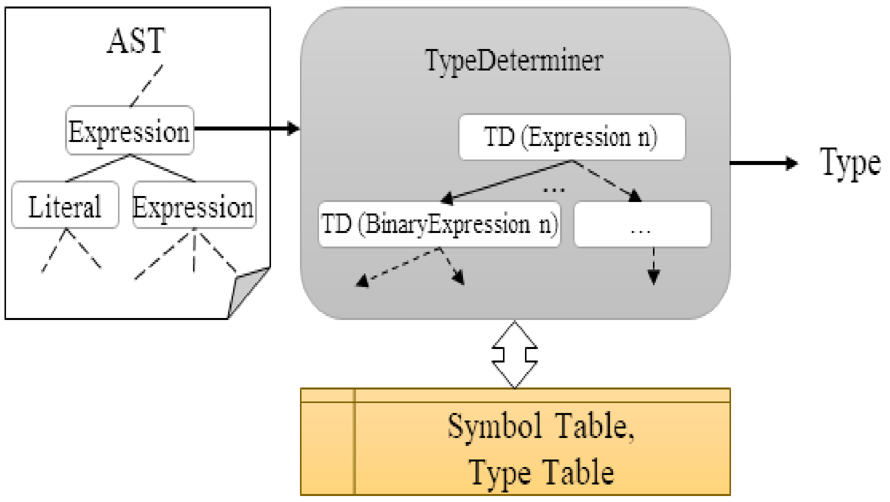


Figure 3. Structure of a Type Analyzer

The type analyzer consists of a set of redundant defined methods that take each AST node that requires type analysis. In order to analyze the type of a given AST, the type analysis method corresponding to the child nodes of the input node is called, and these are collected according to the type analysis rule and are returned.

The types that can be analyzed by the type analyzer are **undefined**, **null**, **boolean**, **string**, **number**, which are basic data types of JavaScript, and **objects**, **arrays**, and **functions** using type information reflected in the symbol table in the semantic analysis stage. In the case of a complex data type, the type analyzer finds and returns a concrete data type from the type information of the type table generated by the declaration processor and the semantic analyzer. The **object** type manages the list of key-value symbols for object property values and associated symbol tables, the **array** type is an array, the function type manages the assembly name of an anonymous function and the object types that can be created when called as a constructor.

3.2.2. Code Generator

The code generator generates SIL, an intermediate language program that is semantically equivalent to the input JavaScript program. The code generator traverses the AST, creates a program with reference to the symbol table and the type analyzer, and calls the type analyzer to check the data type of the code to be output, if necessary.

The code generator consists of a meta information module and a SIL generation module. The meta information creation module writes the contents corresponding to the meta information of the header except the SIL code, the internal symbol table, and the like to complete the assembly file. Figure 4 shows a structure of the code generator.

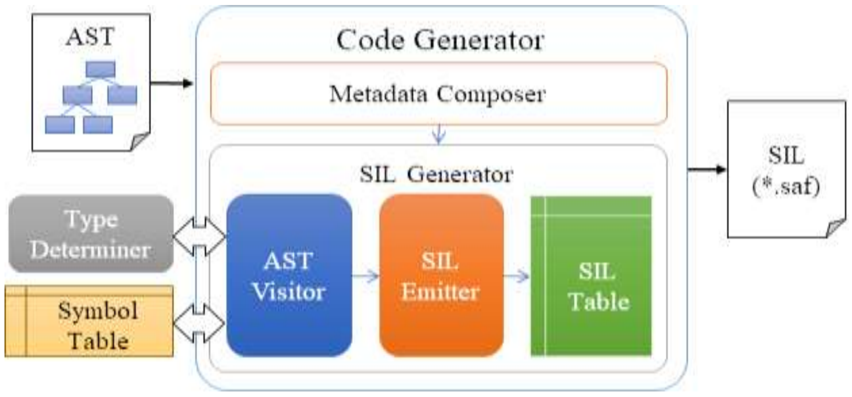


Figure 4. Structure of the Code Generator

The SIL code generation module consists of a visitor module, an emitter module, and table data. The visitor module is implemented as a visitor to the AST and traverses the AST to control the entire code generation flow. Each Visit method calls the Emit method of the Emitter module to generate and aggregate SIL code. In addition to the **Visit** method that implements Visitor, the **VisitConstructor** and **VisitMemberCall** methods are implemented to generate code for the constructor and member function calls. The emitter module generates the SIL code string at a low level and is called from the visitor module. The emitter module consists of a set of emit methods according to the type of SIL code to be generated.

3.2.3. Object-oriented Code Generation

The JavaScript is a prototype-based object-oriented language that accesses the members of the parent object using information from the prototype objects assigned to the object. The JavaScript compiler declare and reference **[[__proto__]]** attributes to compile this like code, If the JavaScript compiler has code to assign objects to prototype properties in the semantic analysis step, add the **[[__proto__]]** property to object symbols on the symbol table to reflect the prototype information, In case of accessing the prototype property when generating code, refer to the **[[__proto__]]** property and output the offset of the object.

The JavaScript does not have a separate constructor, and if you call the function with the new keyword, it will create and return a new object. This means that the meaning depends on whether a particular function is called as a constructor. To accomplish this, the

JavaScript compiler creates a separate intermediate language function with the prefix **&_ctor_** if a particular function is called with the new keyword.

3.2.4. Dynamic Types Support of Virtual Machines

The JavaScript is a dynamic type language in which a data type can be determined at runtime so that the virtual machine can understand the type of the value being executed. To achieve this, SVM is implemented to support basic dynamic types. The dynamic type system SVM (DTS SVM) maintains data types of all values only for the basic data types and array data types of the SIL intermediate language, and can handle cases where data types are changed at execution time. In addition, existing SIL applications are compatible and can run without modification.

All values of the virtual machine have a type index corresponding to the SIL data type, and each value is stored as an address reference to the value in the operation stack and activation record. To reduce the unnecessary memory allocation caused by only storing the address reference in the activation record, the activation record is configured as an offset-value map structure.

Figure 5 shows the DTS SVM value storage structure.

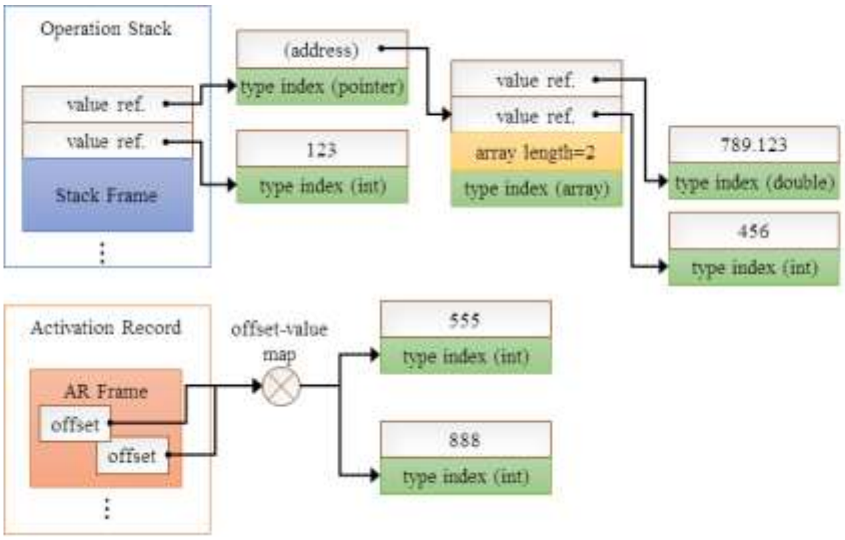


Figure 5. DTS SVM Value Storage Structure

Arithmetic operations between different data types are performed by converting the operands to the appropriate data types during execution using the C language's integral promotion and arithmetic conversion rules. Also, in order to solve the memory problem that arises by operating all the values by the address reference, when the value is loaded from the activation record into the operation stack, it is copied, and when the value is stored, the existing value is deleted and stored.

4. Experimental Results and Analysis

4.1. Type Analysis of Type Analyzer

The type analyzer analyzes and returns the types of expressions and literals that are input based on the analysis data of the previous step, which are used by the semantic analyzer and the code generator when necessary. Figure 6 shows the dump contents of the symbol table as a result of type analysis of the type analyzer of the input source and the respective formulas.

JavaScript Source

```
undefined, null, true, 1+25/3235, "mystring", true?1:"1";
var obj = { prop1: 1, prop2: false };
var fn = function () { return 0; }
```

Type Analysis Result

Input Expression	Analyzed Type
undefined	UndefinedType
null	NullType
true	BooleanType
1+25/3235	NumberType
"mystring"	StringType
true ? 1 : "1"	VarType
{prop1: 1, prop2: false}	ObjectType
function () { return 0; }	FunctionType

Symbol Table Dump

```
# Symbols.SymbolTable : `__global__`
- Kind : ECMASCriptCompiler.Symbols.GlobalSymbolTable
- Outer : null
- Inner.Count : 1
- Symbols
    - VariableSymbol {Name: obj,              Type: ObjectType:
       --objType--1 {DataProperties : 2, AccessorProperties : 0},
       Value: ECMASCriptCompiler.AST.ObjectLiteralExpression}
    - VariableSymbol {Name: fn, Type: FunctionType:
       --fnType--1 {DataProperties : 0, AccessorProperties : 0},
Value: ECMASCriptCompiler.AST.FunctionExpression}
# Symbols.TypeTable
- Types
    - ..bool.. : BooleanType
    - ..null.. : NullType
    - ..number.. : NumberType
    - ..string.. : StringType
    - ..undefined.. : UndefinedType
    - ..var.. : ECMASCriptCompiler.Symbols.VarType
    - --objType--1 : ObjectType: --objType—1
      {DataProperties : 2, AccessorProperties : 0}
    - -objType--2 : FunctionType: --fnType—1
       {DataProperties : 0, AccessorProperties : 0}

# Symbols.ObjectType : `--objType--1`
- Type : ECMASCriptCompiler.Symbols.ObjectType
- Outer : null
- Data Properties
    - Name: prop1,      Type: NumberType,
       Value: LiteralExpression
    - Name: prop2,      Type: BooleanType,
       Value: LiteralExpression
# Symbols.ObjectType : `--fnType--1`
- Type : ECMASCriptCompiler.Symbols.FunctionType
- Outer : null
```

Figure 6. Type Analysis Results and Symbol Table Dump

4.2. Compilation and Execution of Application Programs

In this paper, we designed and implemented the code generator of the JavaScript compiler and implemented basic dynamic type support so that the virtual machine could execute it. Figure 7 shows the JavaScript program that uses the while statement and prime numbers, the SIL intermediate language code that compiled it, and the result of running it in a virtual machine. The JavaScript compiler creates an offset for the object when it references a member of the prototype, and outputs the function as a separate constructor if the called function exists as a constructor. Figure 8 shows the JavaScript program that performs object-oriented inheritance, the code that compiled it into the SIL intermediate language, and the result of the execution by the virtual machine. In the source code, you can see that the intermediate language function **&_ctor_** Class1Constructor corresponding to the function Class1Constructor called as the constructor with new is created.

while.js

```
var j = 0;
while (j < 4) {
   printf("This is while Test. %d %s \n", j, "hi");
   j++;
}
```

prime.js

```
var max = 100;
var i = 0;   var j = 0;
var k = 0;  var rem = 0;
var prime = 0;

i = 2;
while (i <= max) {
   prime = 1;

   k = i / 2;
   j = 2;
   while (j <= k) {
      rem = i % j;
      if (rem == 0) prime = 0;
         j = j + 1;
      }
   if (prime == 1) printf("%d\n", i);
   i = i + 1;
}
```

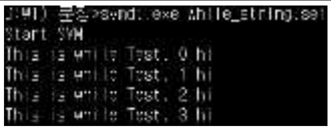
Compiled SIL Code

```
%%HeaderSectionStart
    ...
    .func_name &__entryFunction
    .func_type 1
    .param_count 0
    .opcode_start
       proc 0 0 0
    %Label ##0
       lod.i 0 $j
       ldc.i 4
       lt.i
       fjp ##1
       ldp
       lda 0 @0
       lod.i 0 $j
       lda 0 @1
       calls 40
       lod.i 0 $j
       dup
       ldc.i 1
       add.i
       str.i 0 $j
       ujp ##0
    %Label ##1
       ret
    …
```

Compiled SIL Code

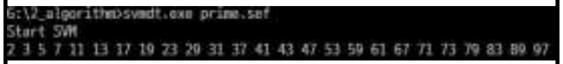
```
%%HeaderSectionStart
     ...
     .func_name &__entryFunction
     .func_type 1
     .param_count 0
     .opcode_start
        proc 0 0 0
        ldc.i 2
        str.i 0 $i
     %Label ##0
        lod.i 0 $i
        lod.i 0 $max
        le.i
        fjp ##1
        ldc.i 1
        str.i 0 $prime
     ...
        fjp ##5
        ldp
        lda 0 @0
        lod.i 0 $i
        calls 40
     %Label ##5
        lod.i 0 $i
        ldc.i 1
        add.i
        str.i 0 $i
        ujp ##0
     %Label ##1
        ret
     .opcode_end
```

Execution Result



```
J:\Y) 로봇>svmd.exe While_string.sef
Start SVM
This is while Test. 0 hi
This is while Test. 1 hi
This is while Test. 2 hi
This is while Test. 3 hi
```

Execution Result



```
G:\2_algorithm>svmdt.exe prime.sef
Start SVM
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Figure 7. Example of the While Statement and Prime Number

inheritancejs

```
var protoObj = {
     a: 10
}
function Class1Constructor() {
     this.c = 20;
}
Class1Constructor.prototype = protoObj;
var obj1 = new Class1Constructor();


printf("\nprotoObj.a = %d\n", protoObj.a);
printf("Class 1 obj1.a = %d\n", obj1.a);
printf("Class 1 obj1.c = %d\n", obj1.c);
protoObj.a = 72;
printf("\nprotoObj.a = %d\n", protoObj.a);
printf("Class 1 obj1.a = %d\n", obj1.a);
printf("Class 1 obj1.c = %d\n", obj1.c);
```

```
%%HeaderSectionStart
    …
    .func_name &__entryFunction
    …
        new_
        call &__ctor__Class1Constructor
        str.p 0 $obj1
        ldp
        lda 0 @0
        lod.p 0 $protoObj
    …
    .func_name &Class1Constructor
    .func_type 0
    .param_count 0
    .opcode_start
         proc 4 1 1
         lod.p 1 0
         ldc.p 0
         add.p
         ldc.i 20
         sti.i
         ret
    .opcode_end
  %FunctionEnd
  %FunctionStart
    .func_name &__ctor__Class1Constructor
    .func_type 0
    .param_count 0
    .opcode_start
         proc 4 1 1
         str.p 1 0
         lod.p 1 0
         ldc.i 4
         cvi.ui
         cvui.p
         add.p
         lod.p 0 $Class1Constructor
      …
```

Execution Result

```
Start SVM


protoObj.a = 10
Class1 Obj1.a = 10
Class1 Obj1.c = 20


protoObj.a = 72
Class1 Obj1.a = 72
Class1 Obj1.c = 20
```

Figure 8. Example of the Inheritance Program

5. Conclusions and Further Researches

In this paper, we designed and implemented a code generator for JavaScript compiler. The code generation process of JavaScript uses a code generator, a type analyzer, and a symbol table that reflects the analysis results of symbols in the previous stage. The type analyzer computes and returns the data types of the expressions based on the AST and symbol table information, and is used to select the appropriate types in the semantic and code generators. The code generator converts the input source code into an intermediate language code having the same meaning by using the AST, the symbol table, and the type analyzer which reflect semantic information by the semantic analyzer. We also implemented a smart virtual machine to support basic dynamic types so that it can execute JavaScript, a dynamic type language. In the future, we plan to extend the SIL intermediate language and virtual machine of smart cross platform to accommodate and support various dynamic type programming languages.

Acknowledgments

This research was supported by Seokyeong University in 2018.

References

[1] D. Crockford, JavaScript: The Good Parts, O'Reilly Media, **(2008)**.
[2] J. Thompsons, JavaScript, Createspace Independent Pub, **(2017**).
[3] Node.js Foundation, Node.js, https://nodejs.org.
[4] Github, Electron, https://electron.atom.io/.
[5] Apache, Cordova, https://cordova.apache.org/.
[6] Y. Lee and Y. Son, "A Study on the Smart Virtual Machine for Executing Virtual Machine Codes on Smart Platforms", International Journal of Smart Home, SERSC, vol. 6, no. 4, **(2012)**, pp. 93-105.
[7] Y. Lee and Y. Son, "A Study on the Smart Virtual Machine for Smart Devices", Information -an International Interdisciplinary Journal, Vol. 16, No. 2, International Information Institute, **(2013)**, Japan, pp. 1465-1472.
[8] S. Han, Y. Son and Y. Lee, "Design and Implementation of the Smart Virtual Machine for Smart Cross Platform", Journal of Korea Multimedia Society, vol. 16, no. 2, **(2013)**, pp. 190-197.
[9] Y. Son, J. Kim and Y. Lee, "Design and Implementation of HTML5 based SVM for Integrating Runtime of Smart Devices and Web Environments", International Journal of Smart Home, vol. 8, no. 3, **(2014)**, pp. 223-234.
[10] Y. Son and Y. Lee, "Smart Virtual Machine Code based Compilers for Supporting Multi Programming Languages in Smart Cross Platform", International Journal of Software Engineering and Its Applications, vol. 8, no. 5, **(2014)**, pp. 249-260.
[11] Y. Son, J. Kim and Y. Lee, "A Study on the JavaScript Compiler for Extension of the Smart Cross Platform", Advanced Science and Technology Letters, vol. 106, **(2016)**, pp. 52-55.
[12] Y. Son, S. Oh and Y. Lee, "A Symbol Table Verification Method for JavaScript Compiler using Reverse Translator on HTML5 Smart Virtual Machine", Information, International Information Institute, vol. 20, no. 5(A), **(2017)**, pp. 5401-5408.
[13] A. B. Tucker and R. E. Noonan, "Programming Languages", 2nd Edition, McGraw-Hill Education, **(2006)**.
[14] D. Grune, H. E. Bal, C. J. H. Jacobs and K. G. Langendoen, "Modern Compiler Design", John Wiley & Sons, **(2000)**.
[15] Y. Son and Y. Lee, "An Objective-C Compiler to Generate Platform-Independent Codes in Smart Device Environments", Information, International Information Institute, vol. 16, no. 2, **(2013)**, pp. 1457-1464.
[16] Y. Son and Y. Lee, "A Study on the Semantic Analyzer of an Objective-C Compiler based on AST-Semantic Tree Transformation", Information, International Information Institute, vol. 17, no. 3, **(2014)**, pp. 1077-1082.
[17] E. Gamma, R. Helm, R. Jhonson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, USA, **(2002)**.
[18] P. Buchlovsky and H. Thielecke, "A Type-theoretic Reconstruction of the Visitor Pattern", Electronic Notes in Theoretical Computer Science, vol. 155, **(2006)**, pp. 309-329.

Authors

JaeHyun Kim, he received the B.S. degree from the Dept. of Mathematics, Hanyang University, Seoul, Korea, in 1986, and M.S. and Ph.D. degrees from Dept. of Statistics, Dongguk University, Seoul, Korea in 1989 and 1996, respectively. He was a chairman of Dept. of Internet Information 2002-2007. Currently, he is a member of the Korean Data & Information Science Society and a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include mobile programming, cloud system and big data analysis.

Yangsun Lee, he received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 1985, and M.S. and Ph.D. degrees from Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 1987 and 2003, respectively. He was a Manager of the Computer Center, Seokyeong University from 1996-2000, a Director of Korea Multimedia Society from 2004-2018, a General Director of Korea Multimedia Society from 2005-2006, a Vice President of Korea Multimedia Society in 2009, and a Senior Vice President of Korea Multimedia Society in 2015-2016. Also, he was a Director of Korea Information Processing Society from 2006-2014 and a President of a Society for the Study of Game at Korea Information Processing Society from 2006-2010. And, he was a Director of HSST from 2014-2018. Currently, he is a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include smart system solutions, programming languages, and embedded systems.