JSSpe: A SYMBOLIC PARTIAL EVALUATOR FOR JAVASCRIPT

by

SÜMEYYE SÜSLÜ

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2018

Her zaman arkamda olan annem ve babama...

To my loving parents Ali and Gülay...

ACKNOWLEDGEMENTS

I would like to gratefully acknowledge Dr. Christoph Csallner for his guidance and support throughout this thesis. His supervision enabled me to acquire numerous skills and gain different points of view to attack the challenges experienced during research. He has also been an excellent mentor. His practical experience and smart solutions have helped and supported me greatly. My thanks extend to respected committee members Dr. David Kung and Dr. Jeff Lei for their feedback and constructive criticism.

I am grateful to my country Republic of Turkey for all of the effort they have put into me even during its unfortunate times. Also, I would like to thank Turkish Ministry of Education for supporting me throughout my Master education.

My fiance Dr. Onur Daşkıran has supported and helped me every time that I need and I'm forever deeply indebted to him.

My loving parents Ali and Gülay, brother Muhammed Enes, sisters Hacer and Emine Merve and my extended family have been intensely supportive and understanding even though I was not with them at times when they needed me around the most.

March 27, 2018

ABSTRACT

JSSpe: A SYMBOLIC PARTIAL EVALUATOR FOR JAVASCRIPT

SÜMEYYE SÜSLÜ, M.S.

The University of Texas at Arlington, 2018

Supervising Professors: Christoph Csallner

Currently, JavaScript is one of the mostly used programming languages for Web and Mobile platforms. This brings a large demand for optimization and smarter resource allocation of the applications written in JavaScript. Partial evaluation is a program transformation technique which rewrites a program by evaluating it with respect to its known variables. Recently, Facebook proposed Prepack: A partial evaluator for JavaScript which will make original program shorter and faster by performing both concrete and symbolic evaluation (concolic evaluation). Although it is proposed as a planned improvement, symbolic evaluation engine currently does not implement an SMT solver. In this work, a JavaScript symbolic partial evaluator (JSSpe) is designed using Babel plugin and it is connected to the Microsoft-Z3 SMT solver to investigate its contribution to its performance. Several test scenarios are experimented in order to show the performance enhancements through using an SMT solver in partial evaluator design.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

Listing Page

CHAPTER 1

Introduction and Motivation

JavaScript is one of the mostly used programming languages for Internet-based Web content productions. As of January 2018, 94.9% of all the websites are employing JavaScript [1]. Almost every modern browsers run with JavaScript. In addition, in recent years, JavaScript has gained more popularity over languages such as Python and PHP in the back-end development. Applications built using JavaScript such as Facebook's React platform serve a large number of users at the same time. This requires a fast execution and efficient usage of resources. Due to this fact it has become crucially important to optimize JavaScript code to make it run faster and consume less resources.

Partial evaluation is a program transformation technique which creates a specialized version of the input program by evaluating all the variables which has known values. It removes unnecessary branches, propagate constants over the program, performs function calls and loop unwinding. This way, it acts like optimizing compilers but it is more powerful as it can create different outputs based on the provided environment. In many JavaScript applications, there are many variables which can be static based on specific scenarios and usually repetitive function calls are performed using this variables. This makes partial evaluation a good candidate for optimizing and specializing JavaScript applications by providing shorter code size and faster execution time.

Original program contains static and dynamic variables in the environment and during execution of partial evaluation these variables can change to dynamic or static and expression of the abstract values can not be done without symbolic execution. Symbolic execution is a technique to execute the whole program without having numerical values of the variables within program. By combining symbolic execution with partial evaluation this restriction can be removed.

Previously, there has been attempts to solve this problem. Jeene [2] is a very limited partial evaluator based on Crockford's parser [3]. Closure compiler [4] optimizes JavaScript code removes dead code and checks JavaScript syntax. Prepack [5] is currently the best solution not only performs concrete evaluation but also implements abstract interpretation. However, it lacks judging satisfiability of abstract expressions. Prepack proposed to solve this problem by incorporating with an SMT solver such as Microsoft-Z3 [6]. However, waiting to be implemented.

In this work, instead of Prepack, JSSpe, a symbolic partial evaluator is created using Babel's [7] plugin and visitor patterns. It is connected to the SMT solver Z3; thus, when it encounters with unsatisfiable abstract expressions, it eliminates using Z3. This is useful especially decision points where the program flow branches different paths. Partial evaluation can determine which branch to take when decision expression is fully concrete but in the abstract case, it should be send to the Z3. JSSpe is a proof of concept implementation; thus, it doesn't cover all the statements. However, experiments performed in this study demonstrate the performance enhancement of combining a symbolic execution engine with a partial evaluation solver.

1.1 Motivating Example

Following example displays a possible pitfall of a partial evaluator without access to an SMT solver. In Listing 1.1, a JavaScript code with three consecutive if-else if statements is shown. The test cases in these if - else if statements are dependent on abstract variable x; thus, they can not be evaluated and omitted using concrete evaluation. However, the second if statement includes an unsatisfiable test condition for all of values of x. A partial evaluator such as Prepack has no connection to an SMT solver. Therefore, it can not remove this unsatisfiable branch from the program and its residual code is shown in Listing 1.2. As can be seen, it keeps branches as it is, renames variables, and does not perform any optimization. JSSpe eliminates unnecessary branches and produces shorter code size and empirically faster run time execution as can be seen in Listing 1.3.

```
var x = Date.now();
function foo(x) {
    var a;
    if (x < 0)
        a = 4;
    else if (x < 2 && x > 8)
        a = 6;
    else if (x > 0)
        a = 8;
    return a;
}
var m = foo(x);
```

Listing 1.1: A Motivating Example: Original foo function

```
  (function () {
var _$1 = this;
var _$0 = _$1.Date.now();
var _1 = _$0 < 0;
var _7 = _$0 < 2;
var _9 = _$0 > 8;
var _6 = _7 && _9;
var _d = _$0 > 0;
var _c = _d ? 8 : void 0;
```

```
  var _5 = _6 ? 6 : _c;
  var _0 = _1 ? 4 : _5;
  m = _0;
}).call(this);
```

Listing 1.2: Residual Code of Prepack

```
    var x = Date.now();
    function foo(x) {
        var a;
        if (x < 0)
            a = 4;
        else if (x > 0)
            a = 8;
        return a;
    }
```

Listing 1.3: Residual Code of JSSpe

CHAPTER 2

Background

Meta-programming is the procedure of generating new programs by analyzing and modifying existing programs [8]. Compilers and interpreters are the most trivial examples of meta-programs since they are programs whose inputs and outputs are also programs. Similarly, partial evaluation is by its nature a meta-programming implementation.

In this chapter, several key concepts on design and analysis of partial evaluators are introduced. Lexical and syntactic structure of programs and ways to represent program syntax in tree form is demonstrated. General definitions of symbolic execution and SMT solvers are explained and through several examples usage of Microsoft-Z3 SMT solver is presented.

2.1 Partial Evaluation

Partial evaluation or also known as program specialization is a technique that creates a specialized program with respect to some of the inputs of an original program which are known at the compilation time. These inputs are also known as concrete or static inputs. A partial evaluator must generate a residual program which runs with remaining inputs and produces same results with the original program which runs with all inputs. Remaining inputs in the residual program are not known at the time of compilation. These are called abstract or dynamic inputs and their values are determined during the execution. Concept of partial evaluation is proven in recursive function theory by Kleene's s-m-n theorem [9]. Inputs of a program can be represented under two categories.

As can be seen in Fig. 2.1, a source program p has two inputs in_1 and in_2 and creates an output. Assuming one of the inputs in_1 has a static value, partial evaluator can transform this source program into a specialized program p_{in_1}, which has only single dynamic input in_2. Partial evaluator is a separate program whose inputs are source code of program p and static variable in_1. Output of the partial evaluator is known as the residual program.

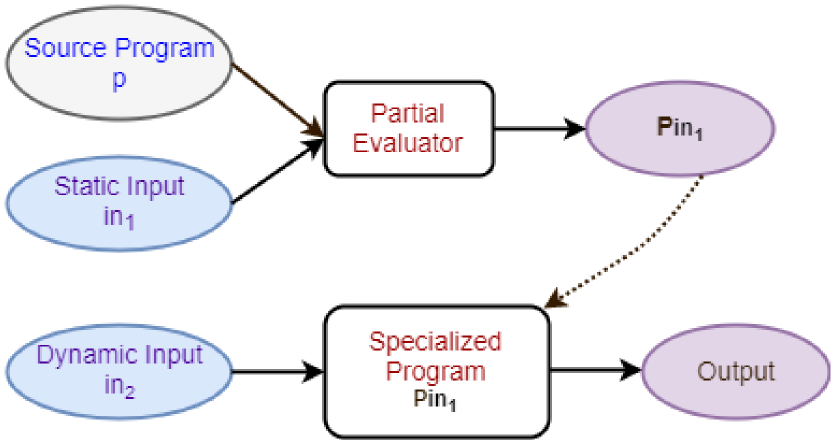


Figure 2.1: General Structure of A Partial Evaluator

Consider following function shown in Listing 2.1 which performs the exponential operation in JavaScript.

```
function power(n,x){
     var p=1;
     while(n>0){
         if(n%2===0){
             x = x*x;
             n = n/2;
         } else{
            p = p*x;
            n = n-1;
        }
    }
  return p;
}
```

Listing 2.1: Original Function

This function has two inputs base, (x) and exponent, (n). In the cases which it is required to compute values for a fixed exponent, this function can be specialized for a static value of exponent using partial evaluation as can be seen in Listing 2.2.

```
function power_3(x){
    return x * x * x;
}
```

Listing 2.2: Residual Function

These two functions create the same output for given base values. Execution of residual function is going to be faster and shorter than the original function as the while loop has been unfolded. On the other hand, this can not be generalized for all cases as partial evaluation does not necessarily guarantee that the specialized program will be any better than the source program [10]. For instance, it is possible to create a specialized program that simply calls the source program with the given static inputs. Even though this specialized program is shorter, it is going to perform worse than the source program due to redundant program calls.

In general, following two conditions should be met in order to talk about the advantages of partial evaluation [10]:

(i) Most computations are based on static data such as library functions are called with constant parameters, in these cases, partial evaluation may have influence on the speed even though no input is provided. This way, greater speed up can be achieved than the optimizing compilers. However, partial evaluators may create excessive amount of loops and code size, hence they are inappropriate as default optimizers.

(ii) These computations are executed more than once, either due to a repetitive structure such as loops or the program itself is required to run several times.

2.1.1 Representation of Partial Evaluation

Partial evaluation can be represented by mathematical equations. In the literature, there are several notations defined for the partial evaluation and throughout this work, notation of [10] is used. In addition, visualizations regarding interpreters, compilers, and Futamura projections are inspired by the work of [11].

A program denotes either the actor which converts inputs into outputs or the textual script which describes the conversion process. These two implications of program should be distinguished. In Fig. 2.2, $[[p]]$ represents the program, while p represents the source code of the program which can be an input or output to other programs. L represents the language which p is written, $x_1, x_2, \ldots, x_n$ can be static or dynamic inputs of the program.

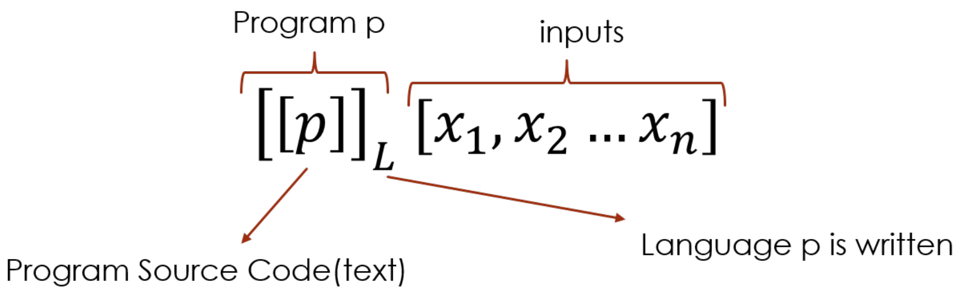


Figure 2.2: Notation For Partial Evaluation

As can be seen in the Eq. (2.1), this notation is used in order to state outputs of original and partially evaluated functions equal each other.

$$[[power]]_{js}\,[3, x] = [[power\_3]]_{js}\,x \tag{2.1}$$

Assume a program p which has two arguments x_1 and x_2. Specializing p with respect to a part of its input $x_1 = d_1$ generates residual program p_{d_1}. This process is performed by another program *peval* and satisfies partial evaluation equation:

$$[[peval]]\,[p, d_1] = p_{d_1} \Rightarrow [[p_{d_1}]]_{d_2} = [[p]]\,[d_1, d_2] \;\; \forall d_2 \tag{2.2}$$

which implies original program p with full inputs d_1, d_2 should yield the same result with residual program p_{d_1} with input d_2.

In Eqs. (2.1)- (2.2), language which the partial evaluator is written (L), as well as the source (S) and target (T) languages are not specified. We assume that all three languages are same for notational simplicity. However, if $L = S$ this opens possibility for applying the partial evaluator itself. For example, Prepack specializes JavaScript code and also written in JavaScript.

2.1.2 Interpreters and Compilers

Program generation procedure of interpreters and compilers can also be represented using a similar notation with the partial evaluation.

2.1.2.1 Interpreters

An interpreter provides a single stage execution. This is also called “run time” and it uses same language with the source program. Operation of the interpreter which is depicted in the Fig. 2.3 must satisfy interpreter equation:

$$[[s]]_S\,d = [[Sint]]_L\,[s, d] \tag{2.3}$$

where s is a source program written in language S, d is input data and S_{int} is an interpreter for language S written in language L. As Eq. (2.3) implies if program s is executed with input d on an S supported platform same result should be obtain when program s and input d are passed to interpreter S_{int} on an L supported machine.

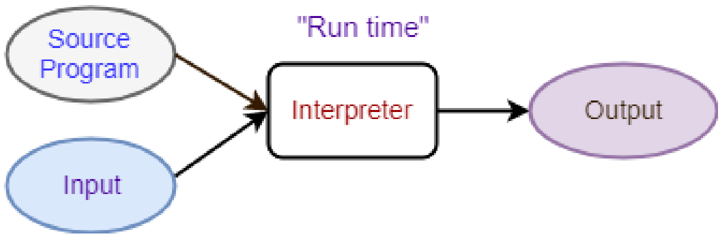


Figure 2.3: Interpreter

2.1.2.2 Compilers

Operation of compilers are performed in two execution stages: "compile time" and "run time" as shown in the Fig. 2.4. A compiler satisfies the equation:

$$[[STcomp]]_L \, p = p' \Rightarrow [[p']]_T \, d = [[p]]_S \, d \;\; \forall d \tag{2.4}$$

where p is a program written in language S, $STcomp$ is a compiler from source S to target T language, and p is the compiled version of program p in L language. Assuming no errors occurring during compilation, executing program p in an L platform with input d should yield the same result when executing program p on a T platform with the same input.

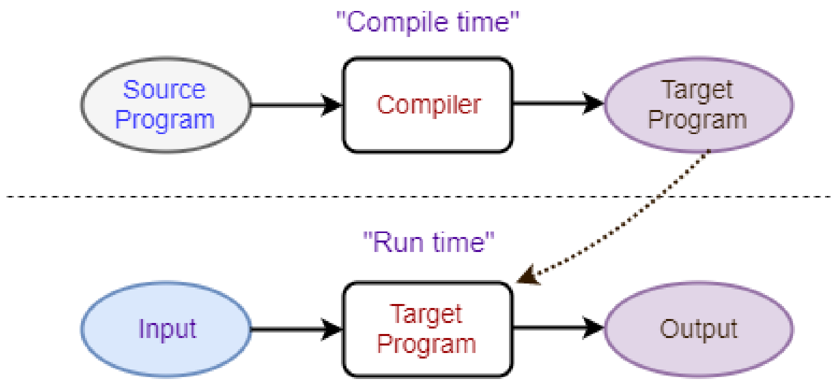


Figure 2.4: Compiler

2.1.3 Compilation Using a Partial Evaluator

A partial evaluator behaves as a crossing between an interpreter and a compiler in the Fig. 2.5.

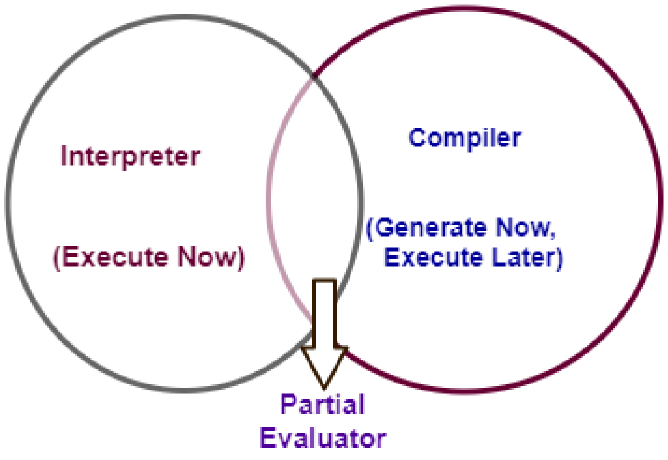


Figure 2.5: Partial Evaluation

If it is expected to execute programs with different inputs in multiple times, it becomes necessary to partially evaluate the interpreter with respect to a fixed known program and unknown inputs. Using partial evaluation, we obtain:

$$[[peval]]\,[S_{int}, s] = S_{int_s} \Rightarrow [[S_{int_s}]]\,d = [[S_{int}]]_L\,[s, d] \quad \forall d \tag{2.5}$$

Using definition of interpreter:

$$[[S_{int_s}]]_L\, d = [[s]]_S\, d \;\; \forall \tag{2.6}$$

Residual program is obtained by compiling source code s from language S into language L. Partial evaluator $peval$ takes two inputs, source code of interpreter S_{int} in L language and source code of s written in S language. It outputs a specialized program S_{int_s} in L language. It creates identical output with program s in S language for a given input d. This shows compilation from S to L is performed using partial evaluation of an interpreter. This is also known as the first Futamura projection and depicted in the Fig. 2.6:

$$[[peval]]\, [interpreter; source] = target \tag{2.7}$$

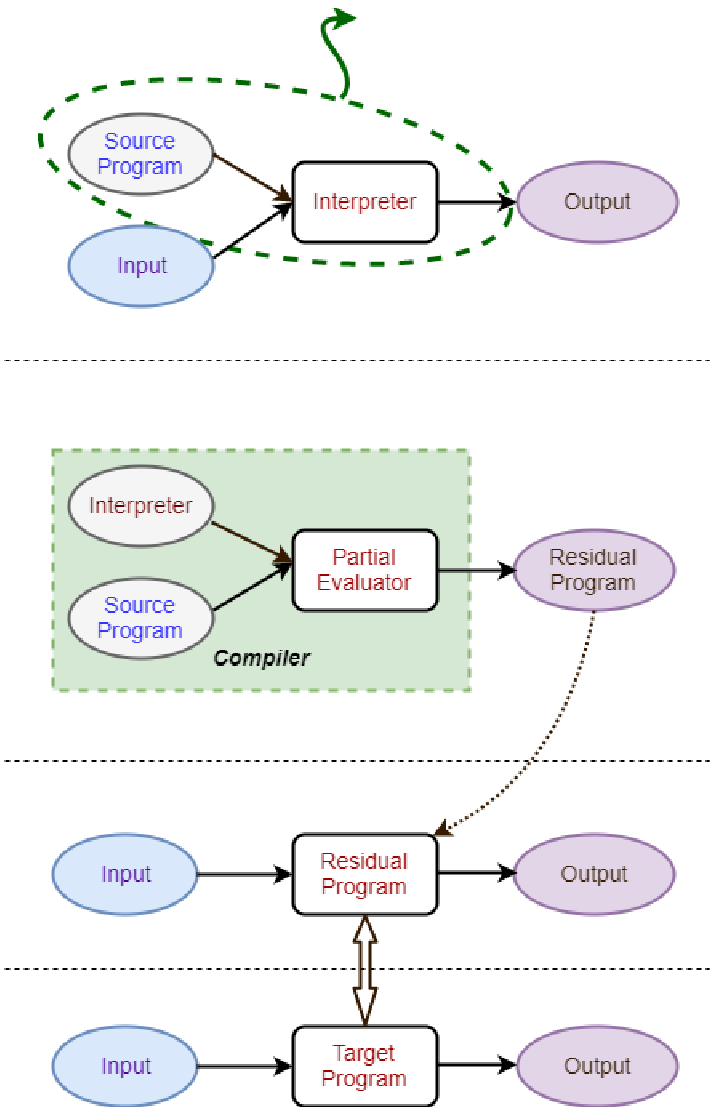


Figure 2.6: First Futamura Projection

Similarly, second Futamura projection, shown in Fig. 2.7 states that a compiler can be generated by partially evaluating an interpreter and a partial evaluator :

$$[[peval]]\ [peval; interpreter] = compiler$$

$$[[compiler]]\ source = target \tag{2.8}$$

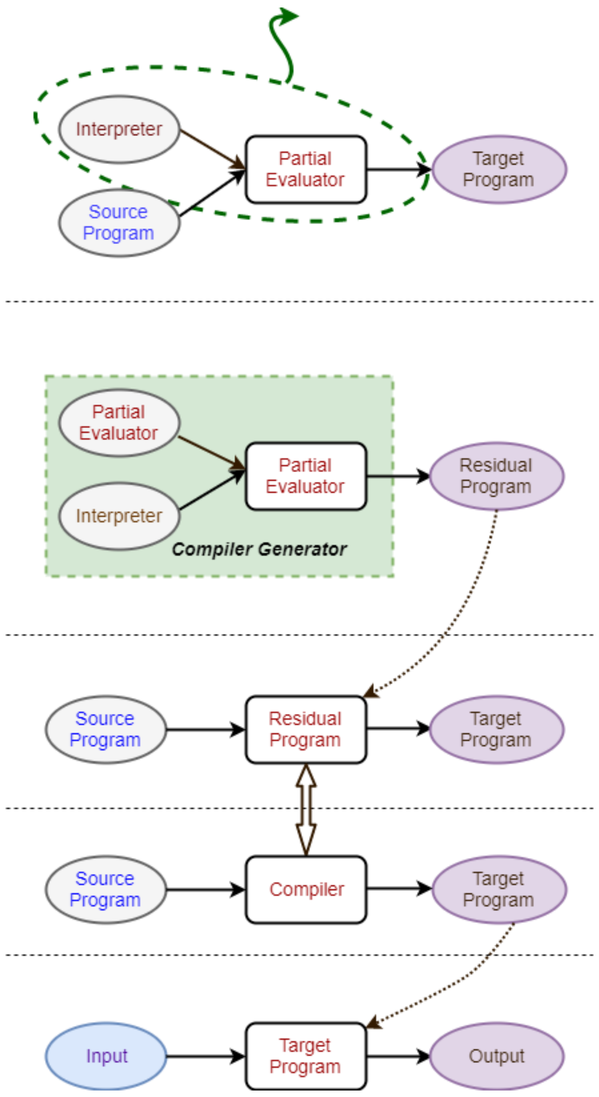


Figure 2.7: Second Futamura Projection

The third Futamura projection depicted in the Fig. 2.8 states that using partial evaluation compiler generator can be generated that is for any given interpreter a compiler is generated:

$$[[peval]]\ [peval; peval] = compilergenerator$$
$$[[compilergenerator]]\ interpreter = compiler \tag{2.9}$$

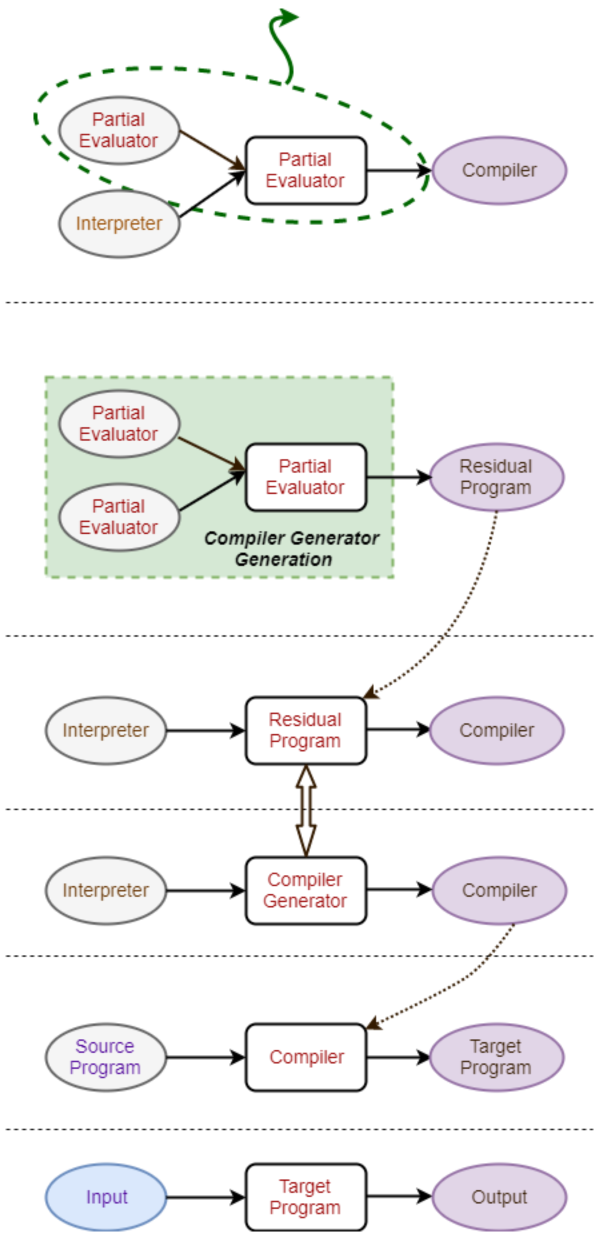


Figure 2.8: Third Futamura Projection

2.1.4 Online versus Offline Partial Evaluation

Based on the analysis and procedure of the partial evaluation stages, partial evaluators are classified into two categories as online and offline.

Online partial evaluators do not perform any analysis of static and dynamic variables. During the partial evaluation, it uses known concrete values of the variables to evaluate expressions and keeps abstract variables as symbolic expressions. In this way, operation of online partial evaluators are quite similar to interpreters. Decisions of concretely evaluating statements or keeping them in the residual code as symbolic expressions are made during the execution of partial evaluator.

On the other hand, offline partial evaluators introduce a preliminary stage where the all variables and expressions in the program are traced and assessed to be either concrete or abstract variables. This procedure is called Binding Time Analysis. Based on the assessed characteristics of the variables, expressions in the program are marked either to be concretely evaluated or to be inserted into residual code. This separation of variables must follow a property known as congruence which states if a concrete variable is updated with an abstract variable, it should be considered as an abstract value. In the second stage, actual partial evaluation is performed and based on the findings of Binding Time Analysis, residual code is created.

2.2 Generation of Abstract Syntax Tree

In the previous section, definition and methods of partial evaluation were presented. Partial evaluators take program source code as their inputs and process them. However, this process uses specialized representations of input programs rather than their textual representation. This type of representations provide partial evaluator to modify, remove or insert new program structures easily. The most common representation used for this purpose is Abstract Syntax Trees (AST).

In this section, first concept of AST is introduced and next a tool for creating and manipulating ASTs in JavaScript language is described.

2.2.1 Abstract Syntax Tree

Abstract syntax tree is a tree representation of hierarchical syntactic structure of source program in a programming language [12]. An abstract syntax tree captures essential structure of the input and operators while omitting unnecessary syntactic details. In the tree data structure of ASTs, interior nodes represent operator and each interior node has children which are operands. In object oriented languages, tree nodes are implemented as classes, instance variables of leaf nodes are used to hold information about nodes value e.g. literal values, references into a simple table.

ASTs can be generated with multiple methods. Hand-written parsers mix parsing code with AST construction code which makes the parser difficult to maintain. Automatic parser generation is an alternative to hand-written parsers, constructing a customized AST data structure is a problem which must be considered [12].

In order to perform necessary operations on nodes of ASTs, all tree structure should be traversed. This can be performed using visitor patterns. In simple terms, visitor patterns are a class definition of methods accepting particular AST nodes [7]. During the AST traversal, each node is examined and if a particular method in the visitor class is encountered, context is passed to the method for execution, so it can be analyzed or manipulated [13].

In the following example, structure of the AST is shown in Fig. 2.9 for a small program is shown in Listing 2.3. AST root node is a *File* node and its children nodes are *Program* and *Comments*. *Comments* node holds static text values of the comment lines existing in the code and this code is not visited as there is no children to it. *Program* node contains all the functional content of the program. All program components make up children nodes based on the program flow. In this example, *Program* node has two children, *Variable Declaration* and *Function Declaration*, respectively. Also, these children nodes have their children. *Variable Declaration* has two children nodes, *Variable Declarator* and *Call Expression*. Subnode of *Variable Declarator* is *id* which is defined as an identifier. *Call Expression* node has two subnodes, *callee* and *arguments*. Type of *callee* node is *identifier* and its name is *myFunction*, arguments node has two *literal values*, 4 and 3. In addition, *Function Declaration* has three children nodes, *id*, *params*, *body*. In the *id* node, function's name *myFunction* is defined as an *identifier*. *params* node has two subnodes and in these subnodes, types of variables are identified and names are determined. Lastly, body node's type is accepted as *Block Statement*, and also *Return Statement* is taken to the AST. *Return Statement* has argument *Binary Expression* and type *Binary Expression Operator* and its subnodes are identified as *left* and *right*. Type of *left* subnode is *identifier* and name is *a*, type of right subnode is *identifier* and name is *b*. It can be stated that during creation of AST, all of nodes are traversing and their names and values are taken.

```
var x = myFunction(4,3);
function myFunction(a,b){
   return a*b;
}
```

Listing 2.3: An Example Source Code

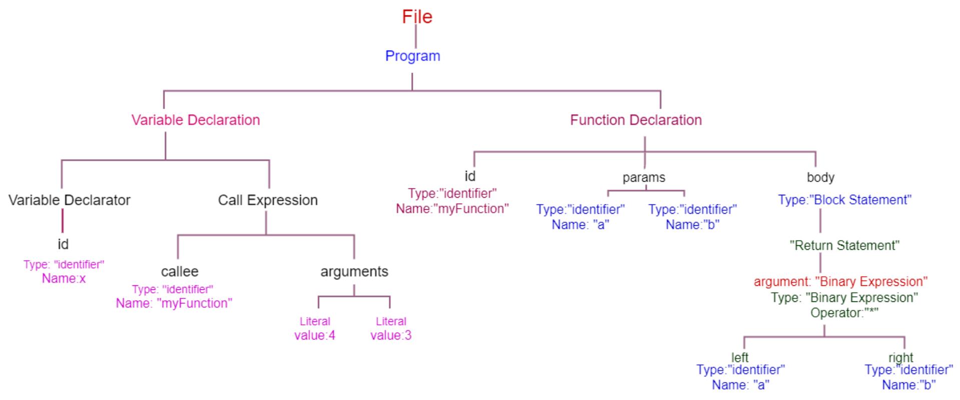


Figure 2.9: Graphical Representation of AST

2.2.2 Babel

Babel is a multi-purpose compiler specifically source to source compiler called as transpiler [7]. It takes a JavaScript source code as input and returns a transformed output source code. This transformation can be generating a code written in the latest ES6 standard into a code which is Browser compatible.

This transformation of the source code is handled using three main tools: parse, transform and generate as can be seen in Fig. 2.10.

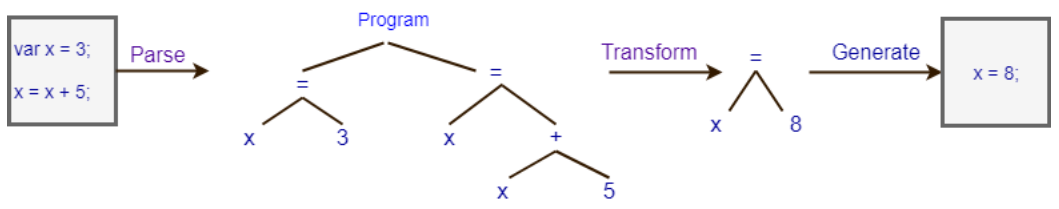


Figure 2.10: Stages of Babel

In the first stage parse, the input source code is converted into an AST. Initially, the source code is read character by character and converted into an array of tokens. This is called Lexical Analysis. In the second part which is called Syntactic Analysis, these tokens are formed in a tree data structure in a way to represent flow of the code.

Main transformations on the AST are performed by traversing all nodes in the tree and implementing specific operations. Babel traverse package provides tools and functions in order to write customized plugin files that can control what sort of modifications can be performed on nodes and connected subnodes. For instance, if a plugin is used to convert ES6 JavaScript code into Browser supported JavaScript code, all visited nodes are replaced with the corresponding ES5 nodes and code transformation is managed without altering the flow of the program. At the end of the process, resulting AST represent desired form with respect to the plugin.

The final stage is performed using Babel generate package. Transformed AST is evaluated once more from top to down and corresponding JavaScript code is generated. Output of this stage is a transformed code and a source map which shows one by one mapping of input code and output code lines. Source maps are mostly used in development purposes such as debugging.

The main transformation of the source code is performed by manipulating the AST by adding, removing and altering its nodes. Nodes in AST are visited Depth-First. These manipulations are provided as plugin files. All nodes of AST are traversed by the plugin and necessary functions are executed in every node based on the types. This is handled by implementing a visitor pattern.

Visitor pattern allows to define a new operation without changing the classes of element on which it operates[14]. During the AST traversal, custom functions are executed whenever specific nodes are encountered. This function can be implemented either as soon as the node is reached which is called $enter()$ method or before leaving the node $exit()$ method. This methods provide flexibility while executing children nodes.

Object representation between nodes is known as path [7]. Path represents the position of node within tree and provides several methods in order to manipulate nodes.

2.3 Symbolic Execution

Symbolic execution is a program analysis technique where the program is executed with provided symbols representing the numerical values of variables used in the program [15]. It has been introduced in mid-1970s for the purpose of creating a tool for software testing. Recently, due to tremendous developments in the computer hardware, it has become a point of interest [16].

In the concrete execution of programs, the numerical values of the variables are fixed. For this reason, only a certain flow graph can be followed during execution. On the other hand, in symbolic execution, multiple control flows can be followed and executed. This provides a faster and more generalized program execution. Symbolic execution is performed using a symbolic execution engine which consist of a symbolic memory store and a model checker [17]. Symbolic memory store holds the symbolic variables and updated as the assignment occur in the path which is executed. Model checker is used to check whether there are any violations on the explored path and constraint solvers are used in the stage.

In the following Listing 2.4, an example JavaScript code which has several if- else if statements is given. All variables in the test condition of the if-else if statements are symbolic variables. A symbolic execution engine visits code line by line and creates a tree representing the execution flow of the code which is called control flow graph (CFG). During the generation of CFG, all possible paths are considered as there is no concrete test statements to decide if a path is taken or not. In Fig. 2.11, control flow graph for the example and procedure of symbolic execution is demonstrated. Symbolic execution traverses all of the nodes and for each decision point, it produces path condition by using the logical representations of the symbolic variables. At the end, constraint solver checks if this path conditions are satisfiable. Based on the response, unsatisfiable paths are eliminated.

```
var x, y, z; // Symbolic
var a = 0; b = 0; c = 0;
if (x > z){
  a = 2;
```

```
    }
    else if (y < x){
      if (z < 3){
          b = -2;
        }
      c = 4;
    }
```

Listing 2.4: An Example JavaScript Code For Symbolic Execution

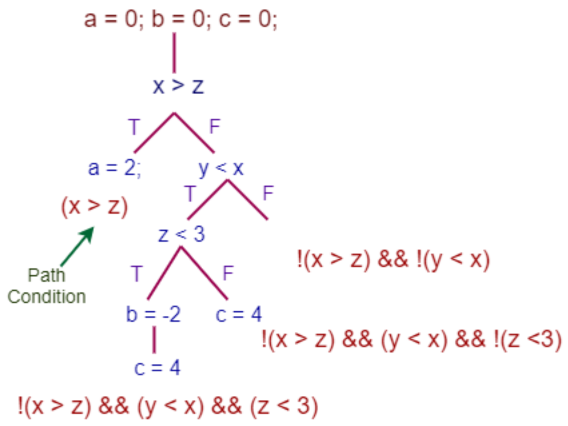


Figure 2.11: An Example of Symbolic Execution Over Control Flow Graph

2.3.1 Constraint Solvers

Constraint resolution is required in different problems such as program analysis, type checking and software testing/verification [18]. Common to all these problems, a solution of a logical expression is searched. Most general version of constraint solving problem is Boolean satisfiability problem (SAT), where a prepositional logic formula with boolean inputs is tested to be satisfied with respect to different combinations of true/false values of its inputs.

SAT solvers implements a systematic search in order to provide a solution to first order logic equations. Formulas are represented in a graph where vertices are the logical variables called *atom*s and the edges are the logical operators such as conjunction, disjunction and negation.

In most of the practical problems, inputs of the constraint formula are not only restricted to Boolean types. Satisfiability Modulo Theories (SMT) are an expansion of SAT, where the inputs of the formula can obtain non-binary types such as integers, real numbers and more complex data structures based on primitive types. SMT solvers employ an abstraction layer where atoms in the formula are converted into logical variables. After the problem is represented in SAT form, SAT solver procedures can be implemented to solve the constraint equations.

In the recent years, attention to the field has increased greatly and many SMT solvers have been introduced. This is partly due to the international SMT competition and the workshop, associated with the annual Computer Aided Verification conference. Novel SMT solvers follow the benchmark and the specifications defined in the SMT-LIB standard. One of the novel SMT solvers developed by Microsoft Research is Z3 and it has been used as the symbolic engine for the efforts of this study.

2.3.2 Microsoft Z3

Z3 is a modern SMT solver which is an active and expressive combination of a large group of solvers [19]. It is developed by Microsoft Research in 2011 and most commonly used to check satisfiability of logical formulas. Z3 is often used as a part of the other tools which performs optimization, sequencing and heuristic search. In this research, Z3 is a component of partial evaluation and symbolic execution.

Z3 has different commands defined in its API that can be executed in the script. In the following, several command types of Z3 is explained with examples.

- All of commands can be accessed typing $(help)$.
- In order to display a message $(echo)$ command can be used.
- $(declare - const)$ defines constant variable of a given type.
- $(declare - fun)$ defines functions.
- Formulas can be added using $(assert)$ command.
- $(check - sat)$ command checks if formula is satisfiable or not for given variables.
- If it is satisfiable, it produces the constant value for given variable and can be accessed using $(get - model)$ command. This command produces the model and explains how the result is created.

In Table 2.1, an example Z3 script with several command types with its corresponding responses are shown. As a result of $(echo)$ command, message inside of the $(echo)$ will be displayed. In addition, type of variable x is integer and written using $(declare - const)$ command. Also, function f has integer and boolean variable type arguments and return type of this function is an integer value. First formula states that value of x variable should be greater than 10. This is expressed using $(assert)$ command. In the second formula, function f must return a result smaller than the 20. In the result of $(check - sat)$ command, Z3 deems the formula satisfiable, thus it can be understood that this formula gives $satisfiable$ result, for at least one value of the variable x. To access one of the these results, $(get - model)$ command is used and value of x is produced as 11.

In addition, if the result of $(check - sat)$ command is $unsat$, this means that for any value assigned to x, it is not possible to satisfy the constraints represented in the formula. In this case, $(get - model)$ command can not be used to access the value

Table 2.1: Usage of Several Z3 Commands For A Satisfiable Example

Several Z3 Commands	Response
`(echo "Z3 is an SMT solver...")`	`Z3 is an SMT solver...`
`(declare-const x Int)`	
`(declare-fun f (Int Bool) Int)`	
`(assert (> x 10))`	
`(assert (< (f x true) 20))`	
`(check-sat)`	`sat`
`(get-model)`	`(model` `(define-fun a () Int 11)` `(define-fun f ((x!0 Int)` `(x!1 Bool)) Int` `(ite (and (= x!0 11)` `(= x!1 true)) 0 0))` `)`

of the variable as can be seen in Table 2.2. It only displays the response as *unsat* since no model can be generated for an unsatisfiable formula.

Table 2.2: Usage of Several Z3 Commands For An Unsatisfiable Z3 Examples

Several Z3 Commands	Response
`(declare-const x Int)`	
`(declare-const y Int)`	
`(assert (= (+ (* 3 x) y) 10))`	
`(assert (= (+ (* 2 x) (* 2 y)) 21))`	
`(check-sat)`	`unsat`

Another point is that if Z3 can not determine whether formula is satisfiable or unsatisfiable, it returns *unknown* as a result.

In some applications, same declarations or formulas may be required to be used again. For this kind of situations, Z3 provides a stack type interface in order to store such declarations and formulas. The command *push* when applied, takes all the commands defined previously and pushes them into the stack and use them in the rest of the program. Similarly, *pop* command works diametrically to the *push* command. It removes all of the declarations or formulas until *pop* command is defined. These commands can be beneficial for saving time and effort.

In the following example 2.3, type of x and y variables is integer and using ($push$) command they are stored into the stack. These variables are applied to the formulas inside of the ($assert$) commands and result is produced as $satisfiable$. Using (pop) command, all of the expressions between *push* and *pop* commands are removed. A new ($push$) command creates a new scope for the new ($assert$) expressions. At this time, $unsatisfiable$ result is produced. Also, new variables can be defined in any place in the code. In this example 2.3, p is defined as Boolean type and using (pop) command, it is definition can be removed and after this process p can not be used without defining it again.

Python, C# and other a few languages have native bindings for Z3 while JavaScript has not. Several workarounds are made in order to combine Z3 features into JavaScript language. One attempt in [20], creates a framework to import Z3 as a Javascript library and implement SMT2 structures by native JavaScript functions. Another proposed solution in [21] first converts expressions into SMT2 language and uses childProcess feature of Node.js in order to create parallel threads which can execute Z3 within the application.

Table 2.3: An Example For Usage of (*push*) and (*pop*) Commands in Z3

Several Z3 Commands	Response
`(declare-const x Int)`	
`(declare-const y Int)`	
`(push)`	
`(assert (= (+ x y) 10))`	
`(assert (= (+ x (* 2 y)) 20))`	
`(check-sat)`	`sat`
`(pop)`	
`(push)`	
`(assert (= (+ (* 3 x) y) 10))`	
`(assert (= (+ (* 2 x) (* 2 y)) 21))`	
`(check-sat)`	`unsat`
`(declare-const p Bool)`	
`(pop)`	
`(assert p)`	`ERROR: unknown constant p`

CHAPTER 3

Overview and Design

In this chapter, a unified solution JSSpe is proposed for the partial evaluation problem using the design principles of partial evaluators and symbolic execution techniques which were described in the previous chapter. In the first section, a simple partial evaluator for JavaScript language (JSSpe) is designed using Babel plugin and combined with Microsoft Z3 to overcome the issues which were mentioned previously. An overview of JSSpe is shown in the Fig. 3.1. At a high level, JSSpe consists of Babel transpiler to convert the JavaScript code into AST, Microsoft Z3 SMT solver to decide if abstract branches are satisfiable or not and the wrapper code to provide interaction between major components.

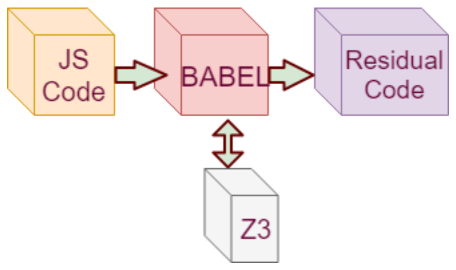


Figure 3.1: JSSpe Workflow Overview

3.1 Partial Evaluator Design

In this section, the tools introduced in the previous chapters are put in use in order to design a partial evaluator in JavaScript language. Partial evaluator is expected to specialize JavaScript programs by elimination of dead code branches, updating and tracking constants, concrete and abstract evaluation of if - else statements, folding of arithmetic and logical expressions, and inlining of function calls.

In the first subsection, the general behavior of partial evaluation is defined using a custom Babel plugin by using a visitor pattern. In the second subsection, the class definitions for converting test structures into SMT language expressions and transferring into Microsoft-Z3 using Child processes in Node.js environment are explained in detail.

3.1.1 Partial Evaluation Using Babel Plugin

Different methods are implemented in nodes of AST in order to partially evaluate a given source code with respect to known concrete and unknown abstract variables.

3.1.1.1 Function Declaration

A function declaration node has several properties which stores name of the function and inputs of the function. During partial evaluation, if one of inputs are known and previously stored in the environment, they are removed from the list of function inputs. This code is located in the *enter*() method since it is desired to remove the known input before it is replaced by its numerical value in children nodes.

```
 FunctionDeclaration: {
  enter(path) {
   for (var i of path.node.params) {
     if (env[i.name] != null && env[i.name].value != null) {
       var index = path.node.params.indexOf(i);
           if (index !== -1) {
             path.node.params.splice(index, 1);
```

Listing 3.1: Function Declaration

3.1.1.2 Variable Declaration

A variable can be declared either as an abstract variable or concrete if it is initialized during declaration. The visitor checks if the variable declaration node owns an *init* property. In this case, a new concrete value is created in the environment and this concrete value is propagated through out the code until it is modified with an abstract value. This is to satisfy constant folding and constant propagating features of a partial evaluator.

```
VariableDeclaration: {
  exit(path) {
      for (var i of path.node.declarations) {
          if (i.init == null) {
              env[i.id.name] = { value: null };
          }
          else if (i.init.type == 'CallExpression') {
              env[i.id.name] = { 'value': null, 'type': "Int" };
              path.node.declarations.splice(path.node.
                 declarations.indexOf(i), 1);
          }
          else {
              if (env[i.id.name] != null
                    && env[i.id.name].value != null)
                  env[i.id.name].value = i.init.value;
              else
                  env[i.id.name] = { value: i.init.value };
          }
          if (path.node.declarations.length == 0) path.remove();
      }
      path.skip();
  }
}
```

Listing 3.2: Variable Declaration

3.1.1.3 Assignment Expression

Assignment expression is inside of the *exit*() method because in the beginning it is desired to evaluate the left hand side of equation. When the evaluation of left hand side is finished, first it is checked whether the left side of the equation is a numerical quantity. In this case, if the variable on right hand side is already registered in the environment its value is updated. Otherwise, a new concrete value is created within the environment. If left hand side is not a numerical quantity, then this means that it is an abstract expression. In this case, if right hand side variable is already registered in the environment, it is removed as it lost its concrete definition. This enables a healthy implementation of constant propagation. At the end of *exit*() method, *path.skip*() function is called in order to skip this newly generated path and avoid possible errors.

```
    AssignmentExpression: {
    exit(path) {
        if (iflvl == 0) {
            if (path.node.right.type == "NumericLiteral"
                || path.node.right.type == "BooleanLiteral") {
                if (env[path.node.left.name] != null) {
                    env[path.node.left.name].value
                        = path.node.right.value;
                    env[path.node.left.name].type
                        = path.node.right.type;
                } else {
                    env[path.node.left.name] = {
```

```
                    value: path.node.right.value
                    };
                }
            }
            else {
                if (env[path.node.left.name] != null) {
                    delete env[path.node.left.name];
                }
            }
            path.skip();
        }
    }
}
```

Listing 3.3: Assignment Expression

3.1.1.4 Logical Expression

In the logical expression node, it is checked whether both operands are concrete boolean literals, then the path is evaluated and whole logical expression node is replaced with a boolean literal node.

```
    LogicalExpression: {
    exit(path) {
        var lval = path.node.left;
        var rval = path.node.right;
        var op = path.node.operator;
        var res;
        if (lval.type == 'BooleanLiteral'
            && rval.type == 'BooleanLiteral') {
            res = path.evaluate();
            path.replaceWith(t.BooleanLiteral(res.value));
        }
        path.skip();
    }
}
```

Listing 3.4: Logical Expression

3.1.1.5 Binary Expression

Similar to logical expression node, in the binary expression node, first thing is checking if both operands are numerical values or identifiers (symbolic variables) whose concrete values are registered in the environment. In these cases, the path can be evaluated and whole binary operation can be replaced with a numerical literal or a boolean literal based on the return type of the operator. However, if only one of operands is known by its concrete value only this operand is replaced with a numerical literal while whole path stays as a binary expression.

```
    BinaryExpression: {
    exit(path) {
      var lval = path.node.left;
      var rval = path.node.right;
      var op = path.node.operator;
      var res;
      if (lval.type == 'NumericLiteral'
                  && rval.type == 'NumericLiteral') {
          res = path.evaluate();
          opPath(res, op, path);
      } else if (lval.type == 'BooleanLiteral'
                  && rval.type == 'BooleanLiteral') {
          res = path.evaluate();
        if (res.value)
```

```
            path.replaceWith(t.BooleanLiteral(true));
      else
            path.replaceWith(t.BooleanLiteral(false));
    } else if (lval.type == 'NumericLiteral'
              && rval.type == 'Identifier') {
      if (env[rval.name] != null && env[rval.name].value != null)
          {
            rval.value = env[rval.name].value;
            path.node.right = t.NumericLiteral(rval.value);
            res = path.evaluate();
            opPath(res, op, path);
      }
    } else if (lval.type == 'Identifier'
              && rval.type == 'NumericLiteral') {
      if (env[lval.name] != null && env[lval.name].value != null)
          {
            lval.value = env[lval.name].value;
            path.node.left = t.NumericLiteral(lval.value);
            res = path.evaluate();
            opPath(res, op, path);
      }
    }
    else if (lval.type == 'Identifier'
              && rval.type == 'Identifier') {
      if (env[lval.name] != null && env[lval.name].value != null
                  && env[rval.name] != null
                      && env[rval.name].value != null) {
            lval.value = env[lval.name].value;
            rval.value = env[rval.name].value;
            path.node.right = t.NumericLiteral(rval.value);
            path.node.left = t.NumericLiteral(lval.value);
            res = path.evaluate();
            opPath(res, op, path);
      } else if (env[lval.name] != null &&
                  env[lval.name].value != null
                        && env[rval.name] == null) {
            path.node.left
                        = t.NumericLiteral(env[lval.name].value);
      } else if (env[lval.name] == null && env[rval.name] != null
                  && env[rval.name].value != null) {
            path.node.right
                        = t.NumericLiteral(env[rval.name].value);
      }
      else if (env[lval.name] != null && env[lval.name].value ==
         null
              && env[rval.name] != null
              && env[rval.name].value != null) {
            path.node.right = t.NumericLiteral(env[rval.name].
               value);
      }
      else if (env[lval.name] != null
                && env[lval.name].value != null
                && env[rval.name] != null
                    && env[rval.name].value == null) {
            path.node.left = t.NumericLiteral(env[lval.name].
               value);
      }
    }
  path.skip();
}
```

Listing 3.5: Binary Expression

3.1.1.6 If Statement

An if statement node has three properties: *test, consequent, alternate*. If *test* passes, *consequent* branch is taken and *alternate* branch is taken in the contrary. Expressions in the *consequent* and *alternate* branches modify the environment in different ways. For this reason, in order to partially evaluate an if statement, it is very important to evaluate the *test* property. In this implementation, two cases for the *test* are considered. First, *test* is assumed to be composed of concrete variables. In the second part, *test* is considered to be an abstract expression which requires symbolic execution.

```
IfStatement: {
enter(path) {
    iflvl++;
},
exit(path) {
    if (path.node.test.type == "BooleanLiteral") {
        if (path.node.test.value == true) {
            if (path.node.consequent.type == "BlockStatement")
               {
                path.replaceWithMultiple(path.node.consequent.
                    body);
            }
            else {
                path.replaceWith(path.node.consequent);
            }
        }
        else {
            if (path.node.alternate != null) {
               if (path.node.alternate.type=="BlockStatement")
                  {
```

```
             path.replaceWithMultiple(path.node.alternate.body);
            }
            else {
                path.replaceWith(path.node.alternate);
            }
        } else
            path.remove();
    }
} else {
    var tmpCode = babel.transformFromAst(t.file(t.program([t.expressionStatement(path.node.test)])));
    var check_SAT = symExec.solvePathConstraint(tmpCode.code);
    if (check_SAT.err) {
        var errorMessage = (check_SAT.err instanceof Error)
            ? check_SAT.err.message
            : 'Uknown error';
        symExec.response.errors.push(errorMessage);
        console.log('error ' + check_SAT.err.message);
    }
    else {
        if (!check_SAT.res.isSAT) { // test unsatisfied,
            console.log('test unsatisfied');
            if (path.node.alternate != null) {
                if (path.node.alternate.type == "BlockStatement") {
                    path.replaceWithMultiple(path.node.alternate.body);
                }
                else {
```

```
                                path.replaceWith(path.node.alternate);
                            }
                        } else
                            path.remove();
                    }
                    else {
                        console.log('test satisfied');
                    }
                }
            }
            iflvl--;
        }
}
```

Listing 3.6: If Statement

3.1.1.6.1 Concrete Branch Decision

Before path is executed, environment is populated with concrete variables through the constant propagation. Both *consequent* and *alternate* branches include expressions such as assignment, binary operation, function calls, etc. and if no special treatment is made, visitor implements these expressions without considering which branch is taken. In the *enter*() method of if statement visitor has not executed children properties; *consequent* and *alternate*. Therefore, the environment is not modified. However, at this point, a decision can not be made, since test is also a children node and it has not been evaluated yet. In the *exit*() method, the test is evaluated and it is possible to make a decision. However, both *consequent* and *alternate* branches are also evaluated and environment has been altered.

One approach is implementing both *enter*() and *exit*() methods in the following way: In the *enter*() method, a flag is raised to avoid execution of assignment operations. This way when children nodes are evaluated, environment is not modified, only *test* is executed. In the *exit*() method, knowledge of which branch is going to be taken is available as well as the environment is still untouched. At this point, the whole if statement node can be replaced with either *consequent* or *alternate* branches depending on the result of *test*. In addition, path is removed completely if no *alternate* is provided and *test* fails. *Consequent* and *alternate* branches may have either single expressions or multiple expressions located in block statement which provides curly braces. For this reason, during the path replacement, this situation is controlled either using *path.replaceWith*() or *path.replaceWithMultiple*() methods. Here, another point to consider is that it is necessary to revisit this new path in order to execute expressions of selected branch and update the environment accordingly. This can be done by removing the *path.skip*() method which is used in the other *exit*() methods.

3.1.1.6.2 Abstract Branch Decision

In this case, same treatment of the concrete decision case is applied in the *enter*() method. In the *exit*() method, if the *test* property has not reduced into a boolean since it includes abstract variables, a direct decision can not be made on which branch to take. In this case, both *consequent* and *alternate* branches are inserted back to the AST. Using *path.skip*() function, it can be avoided to enter the path again and cause the environment to be altered. However, it is also necessary to check all the assignment expressions within both branches and if any variables registered in the environment are used, they need to be removed from the environment as they no longer guaranteed to hold a concrete value.

Although it can not be decided which branch to take by above procedure, in some cases if *test* expression is unsatisfiable, it can be removed from the AST. In order to understand whether an abstract expression is satisfiable or not, an SMT solver such as Microsoft-Z3 can be used. For this purpose, in the cases where the *test* statement is abstract, statement is passed to a symbolic execution function which is detailed in Section 3.1.1.7. According to the result of the symbolic execution, if the *test* statement is not satisfiable under any circumstances, then the *consequent* statement is removed from the AST, and the *alternate* is executed. However, if the symbolic execution returns satisfiable result, no changes are made on the AST since branch is feasible based on the concrete values it will take during run time.

3.1.1.7 Calling Microsoft Z3 From JavaScript

In Section 2.3, Microsoft Z3 and SMT language was explained in detail and several examples were shown. In addition, Z3 API was introduced and usage of Z3 in several languages were described. In this section, conversion process of abstract if-else test conditions into Z3 compatible SMT blocks and general communication architecture between the partial evaluator and Z3 processes are described.

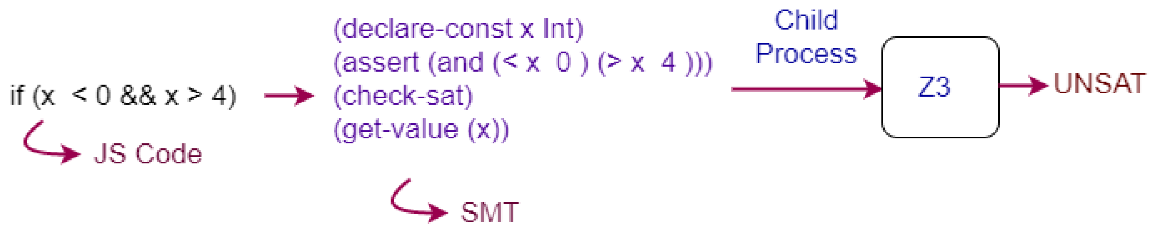
As described in the previous section, partial evaluator program implements the visitor pattern within the Babel plugin and based on the visited node type a different method is executed. During the execution, if partial evaluator decides a branch condition is abstract, it requires Microsoft Z3 to resolve this condition before evaluating the node. All communication, processing and resolving tasks are performed by the methods defined in the *SymbolicExecution* class. This class is partly inspired from a JavaScript symbolic execution engine called Leena [21]. *SymbolicExecution* class consists of a constructor which creates a new symbolic execution environment which requires the valid path of Z3 on the computer and location for a directory to store the SMT files. Main method responsible of solving the constraint is called

$solvePathConstraints()$ and its argument is the text based representation of the expression which is desired to be resolved in the Z3.

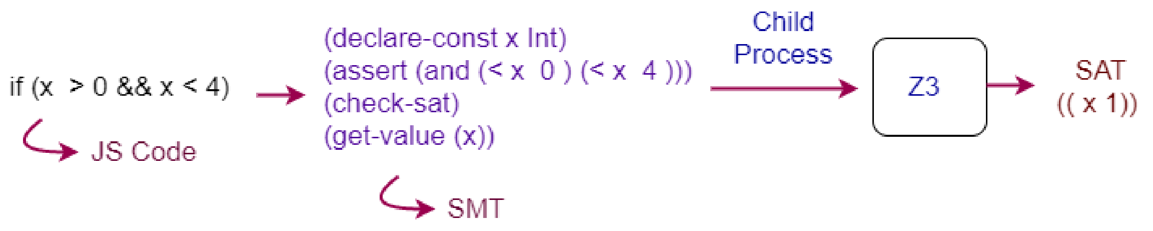
Since in the partial evaluator program, all analysis of the code is performed using the AST representation rather than the textual representation, whenever a condition check is needed, condition cannot be passed to $solvePathConstraints()$ method directly. For this reason, before calling the method, the sub-AST for the condition is embedded into a program AST. This way, a new AST which only contains the test condition is created. This AST is converted into a program source code using Babel's generate function and passed into the symbolic execution method for analysis.

After the test condition is converted into a separate program text and handed over the symbolic execution method, it goes through a parsing process similar to the Babel parse stage using the methods in the $ParserExpression$ class. This parsing process divides the program into its subexpressions and then replaces them with the corresponding SMT language expressions. Following the completion of parsing the program text, some extra lines are inserted in order to display the model generated by Z3 and resulting SMT code is saved to the temporary directory with *.smt2* extension.

Created SMT file can be directly run with Z3 in a console application and computed response can be obtained. Similarly, during the execution of partial evaluator, a new process can be created and the SMT file can be passed into this process. This is performed via the $ChildProcess$ feature of Node.js environment. The $run()$ method of the symbolic execution class creates a new child process and spawns the execution into this new process. Node.js allows to perform running this process both asynchronously and synchronously. For the implementation in this study, a synchronous implementation is preferred in order to maintain compatibility with the other portions of the code. Z3 executes SMT code and determines if the provided condition is satisfiable or unsatisfiable. It also creates an example model if it decides the condition is satisfiable. After the execution of Z3 is completed a response message is created based on the findings of the SMT computation. This procedure is demonstrated in the Fig. 3.2a, for unsatisfiable, in the Fig. 3.2b, for satisfiable branches.



(a)



(b)

Figure 3.2: Stages for the resolution of path conditions using Z3.

The response obtained from Z3 is converted into a string message and sent back to the partial evaluator for further processing. A second parsing process is used to convert this string message into a data structure which carries the information about the result of the satisfiability and the created model if Z3 found the condition to be satisfiable. At this point, partial evaluator uses this data structure to view the satisfiability of the test condition and makes its final decision to either keep or remove the related branch in the AST.

CHAPTER 4

Experiments and Results

In this chapter, first the main objectives of this research effort is described by determining the research questions. Expected results and related hypothesis are defined. Next, test subjects and test environment which was used in the process of execution of the experiments are introduced. Finally, each individual test case is presented and obtained residual code and measured performance metrics are demonstrated.

4.1 Research Questions

To evaluate JSSpe, following research questions are inquired:

(a) Does JSSpe generate a residual program which has faster run time than the original program or residual program obtained by third party partial evaluators such as Prepack?

(b) Does the generated residual program have less line of code than the original program or residual program obtained by third party partial evaluators such as Prepack?

Therefore, the following three research questions (RQ), expectations (E), and hypotheses (H) are investigated:

- RQ1: What is runtime of JSSpe compared to third party partial evaluators?
  - E1: Due to its additional overhead to create parallel threads for evaluating branches in Z3 compared to third party partial evaluators, it is not expected to have faster run time in all cases.

  - H1: JSSpe takes longer time than third party partial evaluators when number of abstract branch expressions are high in the source program.

- RQ2: What is the runtime of residual program generated by JSSpe compared to third party partial evaluators and original program?
  - E2: Assuming the original program has expressions which have the potential to be partially evaluated, it is expected that residual program generated by JSSpe executes faster than the original program as well as the residual program generated by third party partial evaluators. Former is due to evaluation of concrete expressions and modifications (simplify or remove) of symbolic expressions. Latter, depends on the structure of the original program and the amount of symbolic branches which can be removed by Z3.
  - H2: For a program which has potential to be partially evaluated and contains infeasible branches, the runtime of residual program generated by JSSpe is faster than the third party partial evaluators and original program.
- RQ3: What is the code size of residual program generated by JSSpe compared to the residual program generated by third party partial evaluators and original program?
  - E3: Assuming the original program has expressions which have the potential to be partially evaluated, it is expected that residual program generated by JSSpe contains fewer lines of code than the residual program generated by third party partial evaluators and original program.
  - H3: For a program which has potential to be partially evaluated and contains infeasible branches, residual program generated by JSSpe has fewer lines of code than the third party partial evaluators and original program.

4.2 Experimental Setup

Several test subjects are determined in order to adequately answer the research questions put forward by this study. Test subjects are in the form of JavaScript source code which includes different types of expressions. Test cases are passed to both JSSpe and Facebook Prepack version 0.2.19-alpha.0 which is a third party partial evaluator. Based on the obtained residual codes, performances are compared.

For the test runs, JavaScript code is executed in Node.js version 9.2.0 environment on a 6 GB RAM Virtual Machine with Ubuntu 16.04 LTS operating system. Host platform for the Virtual Machine is a Laptop computer with 2.20 GHZ 64 Bit Intel i5-5200 processor, 12 GB RAM and Windows 10 operating system. Measurement of the runtimes are performed using built-in time function in Unix. Both Prepack and JSSpe are executed for 100 times and averaged for each sample cases and using *chrt* mechanism priority of the task is raised to near real time. For the original and residual code runtimes, we used Benchmark.js [22] which is a statistical benchmarking tool for JavaScript. Also, for line of code count, jsmeter [23] is used which counts the statements in the code rather than the line numbers which can be misleading due to different formattings.

4.3 Micro Benchmark: 7 Sample Programs

In this section, several experiment code snippets are generated in order to demonstrate the capabilities of the JSSpe and compare its results with Prepack. These programs include both concrete and abstract statements which partial evaluators either calculate the numerical values or leave as symbolic expressions. Abstract variables are created using Date.now() statement for both partial evaluators. In following subsections results of the experiment cases are shown and discussed.

4.3.1 Concrete Value To Abstract Value

Most of the programs uses several constant parameters and configuration variables. Partial evaluator is expected to use the concrete values of this constants while evaluating the expressions in the program body. For this reason, it is highly important for a partial evaluator to deal with constant variables in a correct and careful manner. Some variables might have constant values in the beginning of the program. However, they can be subjected to operations which converts them into abstract variables. A partial evaluator must be careful to this sort of conversions and update the environment accordingly.

In this experiment, in source code of Listing 4.1, initially, (t) and (rx) are concrete whose values are known and their values are 3, 4, respectively, and variable (q) is abstract whose value is unknown. In Listing 4.2, residual code of Prepack is shown, Prepack computes similar residual code with a different format. In JSSpe, after some binary operations, (t) and (rx) become abstract variables and preserve their abstract values, shown in the Listing 4.3.

```
var k = Date.now();
function foo(k){
  var t = 3, rx = 4;
  rx = rx * k;
  t = 2 + rx;
  return t;
}
var t = foo(k);
```

Listing 4.1: Concrete to Abstract Value (Sample 1).

```
(function (){
  var _$1 = this;
  var _$0 = _$1.Date.now();
```

```
        var _2 = _$0 * 4;
        var _0 = 2 + _2;
        t = _0;
    }).call(this);
```

Listing 4.2: Prepack's Residual Code Creates New Variables to Transfer Current Variables Listing 4.1.

```
    var k = Date.now();
    function foo(k) {
        var t = 3, rx = 4;
        rx = 4 * k;
        t = 2 + rx;
        return t;
    }
    var t = foo(k);
```

Listing 4.3: JSSpe's Residual Code for Listing 4.1.

4.3.2 Concrete Evaluation of If-Else If Statements

In this experiment, in source code of Listing 4.4, our approach is to test concrete values within if-else if statement. Initially, (t) and (rx) variables have values 6 and 2, respectively. Code faces with an if statement and test condition will evaluate according to concrete variable (t). However, the first test condition is already unsatisfiable thus, next test condition will execute. Although second test condition is satisfiable, value of (t) does not satisfy the condition. Since either of the if branches were not taken, they will not affect the environment. Therefore, the last line of code will participate to the execution and produce the residual code. Output of Prepack is shown in Listing 4.5 and output of JSSpe is shown in Listing 4.6 and these outputs are similar because of the concrete evaluation.

```
    var a = Date.now();
    function bar(a) {
        var p, t = 6, rx = 2;
        if (t < 10 && t > 12)
            rx = 6;
        else if (t > 12 && t < 14)
            rx = 8;
        p = a * rx;
        return p;
    }
    var p = bar(a);
```

Listing 4.4: Nested If Statement (Sample 2).

```
  (function () {
  var _$1 = this;
  var _$0 = _$1.Date.now();
  var _0 = _$0 * 2;
  p = _0;
}).call(this);
```

Listing 4.5: Prepack's Residual Code for Listing 4.4.

```
    var a = Date.now();
    function bar(a) {
        var p,
            t = 6,
            rx = 2;
        p = a * 2;
        return p;
    }
    var p = bar(a);
```

Listing 4.6: JSSpe's Residual Code for Listing 4.4.

4.3.3 Abstract Evaluation of If - Else Statements

One of the main contributions of this study is augmenting a partial evaluator with symbolic execution in order to decide when abstract statements are present. If statement is an interesting test case in order to assess the performance of JSSpe.

In Listing 4.7, a simple JavaScript code is demonstrated. Code encounters with an if statement and in order to decide which branch to take, the value of variable (x) should be known. However, (x) is an abstract value and for this reason, partial evaluator can not execute the test condition. Output of Prepack in Listing 4.9 and output of JSSpe in Listing 4.10 produce similar results. However, Prepack creates new variables to transfer current variables and it converts if statement to ternary operation by creating new variables. In JSSpe, when this kind of a situation arises the test statement is passed to the Z3 using an interface to evaluate the branches. The interface code transforms test statement into SMT language as shown in Listing 4.8 and transfers to Z3. In this experiment, test condition is satisfiable thus, during execution of partial evaluator our source code will be kept as it is.

```
var x = Date.now();
function bar(x) {
var a;
  if (x > 0 && x < 2) a = 2; else a = 5;
return a;
}
var a = bar(x);
```

Listing 4.7: Feasible Abstract If-Else Statement (Sample 3).

```
(declare-const x Int)
(assert (and (> x  0 ) (< x  2 )))
(check-sat)
```

```
(get-value (x))
```

Listing 4.8: Z3 Representation of If-Else Statement

```
(function () {
    var _$1 = this;
    var _$0 = _$1.Date.now();
    var _2 = _$0 > 0;
    var _5 = _$0 < 2;
    var _1 = _2 && _5;
    var _0 = _1 ? 2 : 5;
    a = _0;
}).call(this);
```

Listing 4.9: Prepack's Residual Code for Listing 4.7.

```
var x = Date.now();
function bar(x) {
    var a;
    if (x > 0 && x < 2) a = 2;
    else a = 5;
    return a;
}
var a = bar(x);
```

Listing 4.10: JSSpe's Residual Code for Listing 4.7.

Unlike Listing 4.7, in the Listing 4.11, test condition is unsatisfiable and evaluating this branch should be avoided because it will be unnecessary and time-consuming. Prepack produces the result in Listing 4.12 which does not remove unsatisfiable statement and it produces larger code size. JSSpe removes unsatisfiable branch and takes the alternate branch to produce the residual code in Listing 4.13.

In the case our proposed solution is not implemented, partial evaluator can not make a judgment on removing the infeasible if - else statement. The resulting residual code would be the same with the input code in Listing 4.11.

```
var x = Date.now();
function bar (x){
  var a;
  if (x<0 && x>2)
    a=2;
  else if (x>0)
    a=3;
  return a;
}
var a = bar(x);
```

Listing 4.11: An Unfeasible Abstract If-Else Statement Case (Sample 4).

```
(function () {
    var _$1 = this;
    var _$0 = _$1.Date.now();
    var _2 = _$0 < 0;
    var _5 = _$0 > 2;
    var _1 = _2 && _5;
    var _9 = _$0 > 0;
    var _8 = _9 ? 3 : void 0;
    var _0 = _1 ? 2 : _8;
    a = _0;
}).call(this);
```

Listing 4.12: Prepack's Residual Code Does Not Remove Unsatisfiable Statement in Listing 4.11.

```
var x = Date.now();
function bar(x) {
    var a;
    if (x > 0) a = 3;
    return a;
}
var a = bar(x);
```

Listing 4.13: JSSpe's Residual Code for Listing 4.11.

4.3.4 Updating Constant Variables

In a program, there may be several branches and the same constant variables can be modified in different ways in each of these branches. If the partial evaluator can be certain of which branch is going to be taken from concrete evaluation then the values of constant variables are updated in the environment based on the assignment operations in the branch. Source code is in the Listing 4.14. Prepack creates result in Listing 4.15 which does not remove unsatisfiable branch and evaluate the constant variables. On the other hand, JSSpe produces Listing 4.16 which removes unsatisfiable branch and takes care of constant variables.

```
var z = Date.now();
var b = Date.now();
function foo (z, b){
  var x = 5, a = 9;
  if (z>2 && z<0){  b = 5 - a;       }
  else if (x>=5) {  a = 2;           }
  else           {  a = a+2; b = b-1;}
  a = a - 6;
return a;
}
var a = foo(z, b);
```

Listing 4.14: Updating Constant Variables (Sample 5).

```
(function () {
    var _$2 = this;
    var _$0 = _$2.Date.now();
    var _3 = _$0 > 2;
    var _6 = _$0 < 0;
    var _2 = _3 && _6;
    var _1 = _2 ? 9 : 2;
    var _0 = _1 - 6;
    a = _0;
}).call(this);
```

Listing 4.15: Prepack's Residual Code for Listing 4.14.

```
a = -4;
```

Listing 4.16: JSSpe's Residual Code for Listing 4.14.

4.3.5 Folding Binary Operations

Partial evaluation is required to perform several binary operations and retrieve their numerical results if it is possible. It replaces this calculated values with the operation statements and as a result, decisions can be made.

In Listing 4.17, value of (t) is known in the environment and inside of test condition binary comparison operation is performed by first evaluating numerical value of the right side and then evaluating the less than operation. As a result, if statement is removed from the residual code and only its consequent branch is executed which is also a large binary operation. This binary operation is also numerically evaluated to calculate a single $(result)$ value as can be seen in Listing 4.18 and in Listing 4.19.

```
        var k = Date.now();
        function bar(k) {
        var result, t = 2;
        if (t < 1 + 5 * 3) {
            result = 4 + 5 + 6 * 4 * 5 + 5 + 5 / k;
        }
        return result;
    }
    var result = bar(k);
```

Listing 4.17: Folding Binary Operations (Sample 6).

```
  (function () {
    var _$1 = this;
    var _$0 = _$1.Date.now();
    var _2 = 5 / _$0;
    var _0 = 134 + _2;
    result = _0;
}).call(this);
```

Listing 4.18: Prepack's Residual Code for Listing 4.17.

```
    var k = Date.now();
    function bar(k) {
        var result, t = 2;
        result = 134 + 5 / k;
        return result;
    }
    var result = bar(k);
```

Listing 4.19: JSSpe's Residual Code for Listing 4.17.

4.3.6 Abstract While Loops

When the program has an abstract while loop, partial evaluator can not know how many times the loop will be iterated. However, after the while loop ends, it is known that the variables in the test condition are subjected to some constraints even though their actual values are abstract. These constraints can be used to identify if there are any unsatisfiable code structure in the remaining part of the program. Listing 4.20 shows such a condition where the variable a is abstract, whereas it is known to satisfy $a >= 10$ after the while loop ends. Therefore, all code structures using a should also satisfy this constraint and should be removed if not. Currently, Prepack does not support this type of an operation as Listing 4.21 shows the respective error message. On the other hand, JSSpe tracks newly created constraints over the abstract variables and using Z3 decides either to remove or keep the code fragments based on their satisfiability conditions as can be seen in Listing 4.22.

```
var a = foo(Date.now());
function foo (a) {
  var x = 0;
  while ( a < 10){
     a += 1;
    x +=1;
  }
  if (a < 5){
    var y =2;
  }
}
```

Listing 4.20: Abstract While Loops (Sample 7).

```
 This operation is not yet supported on abstract value.
```

Listing 4.21: Prepack's Residual Code for Listing 4.20.

```
var a = foo(Date.now());
function foo(a) {
  var x = 0;
  while (a < 10) {
    a += 1;
    x += 1;
  }
}
```

Listing 4.22: JSSpe's Residual Code for Listing 4.20.

4.4 Evaluation

In this section, research questions are validated based on the results of the experiments. Experiments are performed using the six test cases detailed previously. For all six test cases JSSpe and Prepack are executed in order to partially evaluate the original source codes. Next, the generated residual codes for both JSSpe and Prepack are run individually along with the original source codes. Runtimes of both partial evaluation procedures and the execution of residual programs are measured and experimental data are recorded. Lastly, line of codes in the residual programs and the original program are counted to be compared.

4.4.1 RQ1: Performance of JSSpe and Prepack

Experiment results show that partial evaluation procedure takes around 1160 ms for Prepack and 660 ms for JSSpe. Although this result states that JSSpe has better tool runtime performance, unlike JSSpe, Prepack performs partial evaluation in multiple stages and covers additional evaluation steps. Because of this, comparing JSSpe and Prepack performance solely based on runtimes is not viable. Hypothesis drawn in RQ1 states JSSpe is expected to perform worse based on the fact that creating parallel threads to Microsoft Z3 would lead longer execution times. It is observed that JSSpe takes slightly longer runtimes for the cases which communicate Z3 and the cases that do not.

4.4.2 RQ2: Performance of Residual code

Main performance benefit expected from the partially evaluated code is to run faster since most of the branches are removed. In Sample 1, constant value is propagated through the code. This is expected to lower the number of computations during runtime. Both Prepack and JSSpe show performance benefit. In Sample 2 and 3, both Prepack and JSSpe simplifies the original code.As a result, their runtimes are lower than the original program. In Sample 4, since the if-else statement is feasible, residual code is expected to be unchanged during partial evaluation. Prepack changes the layout of the code to the ternary operations while JSSpe keeps as it is. For this reason, they have similar runtimes. Sample 5 shows benefit of JSSpe as it includes infeasible branches. In Sample 5, Prepack does not remove infeasible branches in the residual code. On the other hand, JSSpe removes infeasible branches and produce shorter residual code. Runtime results show advantage of JSSpe. Sample 6 also demonstrates one of the contributions of JSSpe. Original code has infeasible branches and we want to update value of our variable. In the residual code of Prepack, infeasible branch is not able to be removed and residual code is same as the original code with small layout changes. JSSpe removes infeasible branches and creates a residual code which consists of a single line expression. Therefore, Prepack and original program have slower runtime than the JSSpe. Data of these experiments are displayed on Table 4.1.

4.4.3 RQ3: Comparison of code sizes

Partial evaluator is expected to reduce the size of the original program if there are lines of code which can be concretely evaluated or can be removed as they are infeasible. In cases such as Sample 1 and 4 there are no statements which can result in code removal. Prepack manages to reduce the code size by changing the layout of the code while for JSSpe residual code size stays the same. In Sample 2 and 3 both Prepack and JSSpe shortens the original program expressions. In Sample 5, original code includes an infeasible branch. Residual code of Prepack stays same while JSSpe removes the branch and reduces the code size. In Sample 6, once again, an infeasible branch exists in the original code. Although Prepack can provide some code size reduction by changing layout of the code, JSSpe is able to remove the branch and converts the program into a single line expression. Comparison of code sizes is shown in the Table 4.1.

Table 4.1: Results on micro-benchmark samples: Both JSSpe itself and JSSpe-generated programs had a lower runtime than with Prepack (PP).

	Tool [ms]		Residual [ns]			Residual [LOC]		
#	PP	JSSpe	Orig.	PP	JSSpe	Orig.	PP	JSSpe
1	1160	607	1.51	1.43	1.46	9	6	9
2	1114	652	1.55	1.34	1.30	14	5	8
3	1263	644	1.54	1.41	1.44	9	6	7
4	1213	670	1.65	1.60	1.63	9	8	9
5	1147	842	1.68	1.54	1.51	11	11	8
6	1190	721	1.56	1.54	1.46	17	9	1
7	1103	670	1.71	n/a	1.37	10	n/a	7

CHAPTER 5

Related Work

One of the main objectives of this work is to use partial evaluation in order to increase the performance of JavaScript programs. Several applications used partial evaluation as a tool to improve performance such as speeding up numerical computations [24, 25], decreasing the complexity of rank aggregation problem [26], speeding up execution of GPU compilation [27], removing object allocations and runtime type checks in dynamic languages [28] and improving compilation time and code quality for an embedded instruction set simulator [29].

In recent years, there have been many studies on the symbolic execution of JavaScript programs. Most of these studies aims to find and correct the security vulnerabilities and execution of automatic test cases. One of the earliest of these studies developed a symbolic execution based tool: Kudzu in order to find the vulnerabilities via exploring the execution space of JavaScript programs [30]. Another tool Jalangi provides implementation of heavy-weight dynamic analyses using symbolic execution [31]. It performs a simple taint analysis by implementing concolic testing, and analyzes to track origins of nulls and undefined variables to detect type inconsistencies. A more recent tool SymJS provides a symbolic execution engine for JavaScript, and an automatic event explorer for Web pages [32]. SymJS automatically discovers Web events and symbolically execute the JavaScript code. It produces high coverage automatic test cases based on dynamic feedbacks. In addition to symbolic execution, partial evaluation is proposed as a method for vulnerability detection of JavaScript programs [33].

Although several studies previously considered to combine partial evaluation and symbolic execution, the main objective in these studies is to improve the performance of symbolic execution using partial evaluation. Path explosion problem is a well known problem in symbolic execution especially when the number of paths to be executed is high. A compositional approach to symbolic execution that is based on partial evaluation is proposed to overcome path explosion problem [34]. Symbolic execution capabilities are increased to check liveness properties such as program termination using partial evaluation [35]. Performance boost in Java symbolic execution is demonstrated by introducing a partial evaluator in the design [36]. Same authors presented a software verification system which formalizes Java programming language by capturing sequential semantics [37] and proposed a two stages approach to perform partial evaluation for a Java like language by first symbolically executing the source program then specializing the residual program using symbolic execution tree [38].

Partial evaluation of JavaScript has been proposed in the literature. An online partial evaluator is suggested for improving runtime of browser specific JavaScript code [2]. A more advanced tool aims to shorten JavaScript code size by eliminating inefficient code and generating optimized JavaScript code [4].A recent tool Prophecy improves loading time of JavaScript Web applications by precomputing the data on the server side [39]. Most relevant tool to this study developed by Facebook is Prepack [5]. Prepack performs both concrete and abstract evaluation of JavaScript code in order to generate better runtime performance for JavaScript programs. Prepack currently performs abstract interpretation without employing an SMT solver and it is a planned item for future work to combine it with Z3. JSSpe is inspired from these plans and targets to assess the performance enhancement as a result of integrating a partial evaluator with a Microsoft Z3 SMT solver.

CHAPTER 6

Future Work

Several features of JSSpe can be improved in order to extend its capabilities and coverage. These potential improvements can be performed on the partial evaluation as well as the symbolic execution modules of JSSpe.

As discussed in the previous sections, partial evaluator created for JSSpe aims to provide a proof of concept tool in order to display the possible performance enhancements by combining an SMT solver into a partial evaluator. For this reason, partial evaluator design is only limited for selected source code constructs existing in the AST. A possible enhancement is implementing remaining AST nodes such as for/while loops and recursive function calls in order to cover all program constructs of JavaScript language.

Symbolic execution module of JSSpe can be improved by optimizing the call procedure of Microsoft Z3 and conversion of branch test statements into SMT2 language. In the current implementation, a branch statement is first extracted from AST, converted to a string message which later re-converted into AST during SMT2 conversion. Performance of symbolic execution module might be increased by modifying the current code so that AST structures can directly be passed within functions. In addition, current implementation uses synchronous blocking while starting new threads. Asynchronous callback functions may perform better since they are one of the native features of Node.js.

CHAPTER 7

Conclusions

Partial evaluation is a program transformation technique which rewrites a program by evaluating it with respect to its known variables. Recently, Facebook proposed Prepack: A partial evaluator for JavaScript which will make original program shorter and faster by performing both concrete and symbolic evaluation (concolic evaluation). Although it was proposed as a planned improvement, symbolic evaluation engine did not implement an SMT solver. In this work, a JavaScript symbolic partial evaluator (JSSpe) was designed using Babel plugin and it was connected to the Microsoft-Z3 SMT solver to investigate its contribution to its performance. Several test scenarios were experimented in order to show the performance enhancements through using an SMT solver in partial evaluator design.

REFERENCES

[1] W3Techs, "Usage of javascript for websites," https://w3techs.com/technologies/overview/client_side_language/all, 2018, accessed Jan 2018.

[2] K. Krukow, "Jeene:an automatic partial evaluator for javascript," http://blog.higher-order.net/2008/09/14/jeene.html, 2008, accessed Jan 2018.

[3] D. Crockford, *JavaScript: The Good Parts*. " O'Reilly Media, Inc.", 2008.

[4] Google, "Closure compiler," https://developers.google.com/closure/compiler/, 2015, accessed Jan 2018.

[5] Facebook, "Prepack," https://prepack.io/, 2017, accessed Jan 2018.

[6] Microsoft, "Z3prover/z3," https://github.com/Z3Prover/z3, 2013, accessed Jan 2018.

[7] J. Kyle, "Babel plugin handbook," https://github.com/thejameskyle/babel-handbook/, 2017, accessed Jan 2018.

[8] E. Visser, "Meta-programming with concrete object syntax," in *GPCE*, vol. 2. Springer, 2002, pp. 299–315.

[9] S. C. Kleene, N. de Bruijn, J. de Groot, and A. C. Zaanen, *Introduction to metamathematics*. van Nostrand New York, 1952, vol. 483.

[10] T. Mogensen and P. Sestoft, "Partial evaluation," *Encyclopedia of Computer Science and Technology*, vol. 37, pp. 247–279, 1997.

[11] T. Stuart, "Compilers for free," http://codon.com/compilers-for-free, 2013, accessed Jan 2018.

[12] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Addison-wesley Reading, 2007, vol. 2.

[13] D. Prince, "Understanding asts by building your own babel plugin," https://www.sitepoint.com/understanding-asts-building-babel-plugin/, 2016, accessed Jan 2018.

[14] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," *Reading: Addison-Wesley*, vol. 49, no. 120, p. 11, 1995.

[15] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, July 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[16] I. Erete and A. Orso, "Optimizing constraint solving to better support symbolic execution," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on.* IEEE, 2011, pp. 310–315.

[17] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *arXiv preprint arXiv:1610.00502*, 2016.

[18] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, Sept. 2011. [Online]. Available: http://doi.acm.org/10.1145/1995376.1995394

[19] L. de Moura and N. Bjørner, "Z3-a tutorial," 2011.

[20] B. Loring, "Z3javascript," https://github.com/ExpoSEJS/z3javascript, 2015, accessed Feb 2018.

[21] Mmicu, "Leena," https://github.com/mmicu/leena, 2016, accessed Jan 2018.

[22] M. Bynens and J.-D. Dalton, "Benchmark.js v2.1.2," https://benchmarkjs.com/, 2010, accessed Mar 2018.

[23] N. Peters, "jsmeter - javascript code metrics," http://jsmeter.info, 2016, accessed Mar 2018.

[24] A. Berlin, "Partial evaluation applied to numerical computation," in *Proceedings of the 1990 ACM conference on LISP and functional programming*. ACM, 1990, pp. 139–150.

[25] A. Berlin and D. Weise, "Compiling scientific code using partial evaluation," *Computer*, vol. 23, no. 12, pp. 25–37, 1990.

[26] J. A. Aledo, J. A. Gmez, and A. Rosete, "Partial evaluation in rank aggregation problems," *Computers & Operations Research*, vol. 78, pp. 299 – 304, 2017.

[27] J. Fumero, M. Steuwer, L. Stadler, and C. Dubach, "Just-in-time gpu compilation for interpreted languages with partial evaluation," 2017.

[28] C. F. Bolz, A. Cuni, M. FijaBkowski, M. Leuschel, S. Pedroni, and A. Rigo, "Allocation removal by partial evaluation in a tracing jit," in *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*. ACM, 2011, pp. 43–52.

[29] H. Wagstaff, M. Gould, B. Franke, and N. Topham, "Early partial evaluation in a jit-compiled, retargetable instruction set simulator generated from a high-level architecture description," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*. IEEE, 2013, pp. 1–6.

[30] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 513–528.

[31] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 488–498. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491447

[32] G. Li, E. Andreasen, and I. Ghosh, "Symjs: Automatic symbolic testing of javascript web applications," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 449–459. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635913

[33] O. Tripp, P. Ferrara, and M. Pistoia, "Hybrid security analysis of web javascript code via dynamic partial evaluation," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 49–59. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610385

[34] J. M. Rojas and C. S. Pasareanu, "Compositional symbolic execution through program specialization," *BYTECODE'13 (ETAPS)*, 2013.

[35] G. Vidal, "Closed symbolic execution for verifying program termination," in *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*. IEEE, 2012, pp. 34–43.

[36] R. Bubel, R. Hähnle, and R. Ji, "Interleaving symbolic execution and partial evaluation," in *Formal Methods for Components and Objects*. Springer, 2010, pp. 125–146.

[37] R. Bubel, R. Hähnle, and R. Ji, "Program specialization via a software verification tool," in *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO)*. Springer, Nov. 2010, pp. 80–101.

[38] R. Ji and R. Bubel, "PE-KeY: A partial evaluator for java programs," in *Proc. 9th International Conference on Integrated Formal Methods (IFM)*. Springer, June 2012, pp. 283–295.

[39] R. Netravali and J. Mickens, "Prophecy: Accelerating mobile page loads using final-state write logs," in *Proc. 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2018.

BIOGRAPHICAL STATEMENT

Sumeyye Suslu was born in Kayseri, Turkey in 1991. She received her B.Sc. degree in Computer Science from Gazi University, Turkey, in 2013. Following that, she started her Master in Program Curriculum in Education in Gazi University and continuing it. After that, she started her M.Sc program in Software Engineering major in Computer Science and Engineering department in The University of Texas at Arlington in 2016. She was also appointed as Graduate Teaching Assistant (UTA) one semester in UTA. Her research interests are JavaScript, symbolic execution, SMT solvers, software testing, software measurement and quality.