Durham E-Theses

Verification and Validation of JavaScript

XIONG, WEI

How to cite:

XIONG, WEI (2013) *Verification and Validation of JavaScript*, Durham theses, Durham University. Available at Durham E-Theses Online: http://etheses.dur.ac.uk/7326/

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, Durham University, University Office, Old Elvet, Durham DH1 3HP
e-mail: e-theses.admin@dur.ac.uk Tel: +44 0191 334 6107
http://etheses.dur.ac.uk

Verification and Validation of JavaScript

Wei Xiong

School of Engineering and Computing Sciences

Durham University

A thesis submitted for the degree of

Doctor of Philosophy

2012

I would like to dedicate this thesis to my loving parents,

for their concern and precious support.

Acknowledgements

I would like to give deep appreciation to the people who have helped me in the years for this thesis. First of all, it is great privilege to work with my supervisor Emeritus Professor Malcolm Munro, his invaluable guidance and proficient knowledge on our research always benefited me.

Secondly, I'd like to thank to my former supervisor Professor Shengchao Qin who initiated my research direction and provided technical support. This thesis will not able to be finished without their supervision.

Thirdly, my sincere thanks to my colleagues, Chenguang Luo and Guanhua He, their knowledge over program verification and precious discussions with them really inspired our research. The continuous encouragement from other colleagues gave me more confidence to constantly work hard on research, they are Aziem Chawdhary, Chris Watson, Jamie Godwin, Granville Barnett, Ryuta Arisaka, and the people who I have been worked with in Durham University.

In addition, I would also need to thank Professor Zongyan Qiu and Dr Huibiao Zhu for sharing their knowledge and research experience on program formalisation and language semantics. They helped me explore my experience on separation logic and operational semantics.

I also want to acknowledge Ander Moller for sharing his experience on JavaScript type analysis system, a senior JavaScript architect Douglas Crockford for his encouragement on developing a formal verification system for the JavaScript,

Dr. Sergio Maffeis for sharing his research experience on designing a program logic for the JavaScript.

Finally, I could not thank enough to my parents. Their constant love, concern, encouragement, and patience they have given me through my life kept me staying positive and went through the course of this thesis.

Abstract

JavaScript is a prototype-based, dynamically typed language with scope chains and higher-order functions. Third party web applications embedded in web pages rely on JavaScript to run inside every browser. Because of its dynamic nature, a JavaScript program is easily exploited by malicious manipulations and safety breach attacks. Therefore, it is highly desirable when developing a JavaScript application to be able to verify that it meets its expected specification and that it is safe. One of the challenges in achieving this objective is that it is hard to statically keep track of the heap-manipulating JavaScript program due to the mutability of data structures. This thesis focuses on developing a verification framework for both functional correctness and safety of JavaScript programs that involve heap-based data structures.

Two automated inference-based verification frameworks are constructed based upon a variant of separation logic. The first framework defines a suitable subset of JavaScript, together with a set of operational semantics rules, a specification language and a set of inference rules. Furthermore, an axiomatic framework is presented to discover both pre/post-conditions of a JavaScript program. Hoare-style specification **{Pre}prog{Post}**, where program **prog** contains the language statements. The problem of verifying program can be reduced to the problem of proving that the execution of the statements meets the derived specification language.

The second framework increases the expressiveness of the subset language to

include **this** that can cause safety issues in JavaScript programs. It revises the operational rules and inference rules to manipulate the newly added feature. Furthermore, a safety verification algorithm is defined.

Both verification frameworks have been proved sound, and the results obtained from evaluations validate the feasibility and precision of proposed approaches. The outcomes of this thesis confirm that it is possible to analyse heap-manipulating JavaScript programs automatically and precisely to discover unsafe programs.

Copyright © 2012 by WEI XIONG.

The copyright of this thesis rests with the author. No quotations from it should be published without the authors prior written consent and information derived from it should be acknowledged

Contents

1 Introduction 1

1.1 Background . . . 2

1.2 Motivation . . . 5

1.3 Objectives . . . 6

1.4 Criteria for Success . . . 7

1.5 Thesis Organization . . . 9

2 Literature Review 11

2.1 Introduction . . . 11

2.2 JavaScript as a Scripting Language . . . 12

2.2.1 Main Features of JavaScript . . . 14

2.2.1.1 Data Type and Variable . . . 14

2.2.1.2 Object Structure . . . 15

2.2.1.3 Functions . . . 15

2.2.1.4 Prototype and Inheritance . . . 17

2.2.1.5 Other Conventions . . . 22

2.2.2 Client-Side JavaScript . . . 25

2.2.3 ADsafe . . . 30

2.2.4 Summary . . . 33

2.3 Formal Verification . . . 34

2.3.1 Model Checking 36
2.3.2 Hoare Logic and Verification 37
2.3.3 Separation Logic and Verification 39
2.3.4 JavaScript Program with Formal Framework 41
2.4 Summary 43

3 JS_{sl} - A Subset of JavaScript **47**
3.1 Introduction 47
3.2 The Language JS_{sl} 48
3.2.1 The Features and Conventions of JS_{sl} 49
3.2.2 The Syntax of JS_{sl} 51
3.3 Example 53
3.4 Semantics for JS_{sl} 55
3.4.1 Semantic Domain 55
3.4.2 Operational Semantics 59
3.5 An Axiomatic Framework for JS_{sl} 68
3.5.1 Specification Language for JS_{sl} 70
3.5.2 Inference Rules 72
3.5.3 Soundness 81
3.6 Summary 83

4 ${JS^t}_{sl}$ - A safe usage of this for JS_{sl} **84**
4.1 Introduction 84
4.2 Example Analysis 87
4.3 Reachability Graph Analysis for ${JS^t}_{sl}$ 95
4.4 Verification 99
4.4.1 The Language ${JS^t}_{sl}$ 99
4.4.2 Revised Operational Semantic Rules 101

4.4.3 Specification Language for $JS^t{}_{sl}$ . . . 102
4.4.4 Main Verification Algorithm . . . 105
4.4.5 Formal Property: Safety . . . 107
4.4.6 Revised Inference Rules . . . 109
4.4.7 Soundness . . . 114
4.5 Summary . . . 114

5 Case Studies and Evaluation **115**
5.1 Introduction . . . 115
5.2 Case Studies . . . 115
5.2.1 Case Study A . . . 117
5.2.2 Case Study B . . . 123
5.2.3 Case Study C . . . 129
5.2.4 Case Study D . . . 134
5.3 Evaluation . . . 137
5.3.1 Analysis of Case Studies . . . 137
5.3.2 Evaluation of $JS^t{}_{sl}$ Framework . . . 139
5.4 Summary . . . 142

6 Conclusion **143**
6.1 Introduction . . . 143
6.2 Contribution . . . 144
6.3 Criteria for Success . . . 145
6.4 Future Work . . . 147
6.5 Summary . . . 148

A Appdx A **149**

B Appdx B **158**

References 182

List of Figures

1.1 Thesis Structure . . . 10

2.1 Three Important Concepts in JavaScript . . . 14
2.2 Variables Declaration in JavaScript . . . 16
2.3 Objects Structure in JavaScript . . . 16
2.4 Closure in JavaScript . . . 18
2.5 Function Declaration vs. Function Expression . . . 18
2.6 Inheritance in JavaScript . . . 19
2.7 Inheritance of Function Object in JavaScript . . . 19
2.8 Metaprogramming in JavaScript . . . 23
2.9 Overview of Web Browser Implementation . . . 24
2.10 ADsafe Structure in HTML Document . . . 32
2.11 Frame Rule in Separation Logic . . . 41

3.1 JavaScript v.s. JS_{sl} . . . 48
3.2 Syntax of JS_{sl} . . . 54
3.3 JavaScript Example . . . 56
3.4 JS_{sl} Example . . . 56
3.5 Operational Semantics for Variable Assignments . . . 61
3.6 Operational Semantics for Field Statements . . . 62

3.7 Operational Semantics for Function Invocation 65
3.8 Operational Semantics for Object Creation 67
3.9 Operational Semantics for Control Structures 69
3.10 The Specification Language $Spec_{sl}$ 72
3.11 The Semantic Model for $Spec_{sl}$ 73
3.12 The Semantic Model for Pure Formula in $Spec_{sl}$ 74
3.13 Inference Rules for Variable Assignments and Field Statements 76
3.14 Inference Rules for Function Invocation 78
3.15 Inference Rules for Object Creation 80
3.16 Inference Rules for Control Structures 80

4.1 Example of **this** variable manipulation in JavaScript 86
4.2 Example of **this** variable manipulation in ${JS^t}_{sl}$ 88
4.3 Reachability Graph Notations 89
4.4 Reachability graph after line 1 90
4.5 Reachability graph after line 6 90
4.6 Reachability graph after line 8 91
4.7 Reachability graph after line 10 92
4.8 Reachability graph after line 12 93
4.9 Reachability graph after line 14 94
4.10 Reachability graph after line 16 96
4.11 Reachability graph after line 18 97
4.12 Syntax of ${JS^t}_{sl}$ 100
4.13 Revised [**op-mutate-field**] Rule 101
4.14 The Specification Language ${Spec^t}_{sl}$ 103
4.15 Scope and Prototype Lookup Chain 104
4.16 The Additional Semantic Model for ${Spec^t}_{sl}$ 105

4.17 Logic Predicate Semantic Model for $Spec^t{}_{sl}$ 110
4.18 Updated Inference Rules for Variable Assignments and Field Statements . 111
4.19 Updated Inference Rules for Function Invocation 112
4.20 Updated Inference Rules for Object Creation 113
4.21 Updated Inference Rules for Control Structures 113

5.1 JavaScript for Case Study A 118
5.2 $JS^t{}_{sl}$ for Case Study A (Phase 1) 118
5.3 Program Analysis for Case Study A (Phase 2) 120
5.4 Algorithm Application for Case Study A (Phase 3) 122
5.5 JavaScript for Case Study B 124
5.6 $JS^t{}_{sl}$ for Case Study B (Phase 1) 124
5.7 Program Analysis for Case Study B (Phase 2) 126
5.8 Algorithm Application for Case Study B (Phase 3) 128
5.9 JavaScript for Case Study C 129
5.10 $JS^t{}_{sl}$ for Case Study C (Phase 1) 130
5.11 Program Analysis for Case Study C (Phase 2) 132
5.12 Algorithm Application for Case Study C (Phase 3) 133
5.13 JavaScript for Case Study D 134
5.14 $JS^t{}_{sl}$ for Case Study D (Phase 1) 135
5.15 Program Analysis for Case Study D (Phase 2) 136
5.16 Algorithm Application for Case Study D (Phase 3) 137

List of Tables

2.1 Overview Comparison Between JavaScript and Java 13

2.2 Dot prototype vs. Dot [[prototype]] 21

2.3 Comparisons of Frameworks for JavaScript 46

4.1 Features Comparison of JS_{sl} and JS^{t}_{sl} 101

4.2 Properties Comparison of $Spec_{sl}$ and $Spec^{t}_{sl}$ 102

5.1 Summary of Case Studies 139

5.2 JS^{t}_{sl} vs. other frameworks 141

List of Algorithms

1 Safe_This Algorithm 107

Chapter 1

Introduction

Computers were developed in the last century and the computer revolution has grown to have a huge impact on peoples' daily lives, computer-based systems play an increasingly significant role across all aspects of human life. People heavily rely on various web technologies, such as social networking, online shopping, and email. Client-side web technologies allow users to participate more interactively and collaborate with each other. For example, a scripting language like JavaScript can be used to upload and download data from a web server without undergoing a full page reload. However, JavaScript is double-edged sword. The greater power of the language also brings a greater risk. JavaScript lacks the maturity of the object-oriented languages, for example, every block of code shares the same execution priority, the private variables and methods can be accessed by an outside object, the same code can produce different output under different browsers. Therefore, the users' private information and the authority of the web pages could be breached because of a malicious client-side JavaScript application.

Web applications provide the potential for malicious parties to deliver scripts to run on a client computer via the web, so that the web registered users' private information can be leaked if the web page was compromised or under attack. For example, the social networking website LinkedIn announced that 6.5 million of its members' passwords were

posted onto the Internet after the website was compromised on June 2012 (Lin12). The users' information were leaked through a breach of *log in* JavaScript application that connects to database. Another example, a malicious third party advertisement exploited The New York Times web site (Newch) in 2009. The New York Times website included some advertisements from third party advertisers' servers which changed the advertisement to take over the entire window/web page (host page) and entice users into downloading fake anti-virus software.

Since unsafe web applications can bring catastrophic results, a systematic method is urgently needed to ensure that the developed applications are safe, robust and fulfill the requirements of users. However, the growth of web applications in terms of size and complexity make checking this manually almost impossible. Besides, testing is limited in its test sets to discover latent threats (Dij72). A solution is to develop a framework based on formal verification approach to verify software, and enhance the quality of web applications (HM04). The aim of this thesis is to provide a formal method to enhance the safety of web applications.

1.1 Background

There are various approaches to improve the quality of software, including program language design, formal specification language design, program testing, model checking, static analysis and other program analysis techniques and verification.

A design for programming language can avoid certain kinds of programming errors (JMG+02; CHA+07). For example, the Java language adopted the design of a garbage collector which eliminates the needs of explicit memory allocation and deallocation that can prevent program execution from heap space memory exhaustion (Ven99). Another example is the design of Haskell and OCaml, which eliminate type casting to prevent type errors from occurring at run time (Jon03; LDG+10). Google developed the Dart language

to replace JavaScript for enhancing the safety and security of scripting languages (Darge). However, such an approach cannot guarantee the correctness of programs, and it also cannot remove the issues that already exist in current languages.

Formal specification is another approach to enhance the quality of programs. It involves using a specification language to specify the requirement and behaviour of a program in a rigorous mathematical way. The specification language is usually a higher level language than the programming language. It creates a foundation during program analysis and verification to specify the programs for further analysis, such as correctness analysis of the programs. For example, UML was designed to model applications (Jac99), CSP describes patterns of interaction in concurrent system as a formal specification language (Hoa78), CASL is a Common Algebraic Specification language based on first-order logic with induction (BM04), and the Z specification language is based on typed first-order predicate logic and produced to describe complex dynamic systems in a smaller and simpler mathematical way than any programming language can provide (Spi89).

Program testing is a another widely applied and efficient approach to provide information about the reliability and quality of software. It is a process of validating and verifying that the software works as expected or satisfies the needs of stakeholders by executing programs with certain sets of input values and determining whether the software can satisfy the input-ouput specification. The challenge for software testing is in automatically generating a test set to a high standard (Bez90; HH91). The complexity of the software makes the completeness of the infinite test cases set impossible (Dij72). Model-based testing (MBT) provides an approach to analyse the test results instead of focusing on the test set generation (Liu11). This approach proposes a specification based upon a formal model to generate expected inputs and outputs. It compares the actual generated outputs with expected outputs, and is able to decide on further actions such as modifying the model and generating more tests, or stopping testing and estimating reliability of the software. However, verifying the completeness of the test sets

is a challenge in software testing. The test sets may improve the degree of the precision and efficiency of the approach, but it cannot cover certain explicit characteristic of the program, such as functional correctness and safety.

To overcome the weaknesses of testing, formal verification is proposed to be another path ensuring the qualify of software. It uses mathematical based approaches to perform the static analysis and verification of programs. Model checking is introduced to automatically verify the correctness of finite-state systems by an exhaustive exploration of the space of the computation states according to a specification in temporal logic (CE81; QS82). It uses abstraction techniques to calculate potentially infinite sets of computation states to finite states. The SLAM [1] model was successfully applied to Microsoft Static Drive Verifier (BR02; BCLR04). BLAST is a C language program model checker that employs counterexample guided automatic abstraction refinement (HJMS03).

Another formal verification approach is deductive logic inference verification. It uses logic formulas to formally describe the behaviour of programs and interpret program statements as predicate transformers to reason about program with axioms and inference rules in axiomatic systems (Flo67; Hoa69).

Static program analysis is used to derive properties of programs in a systematic way. In contrast to dynamic analysis, it executes program code symbolically based on an abstract model (NNH99). The abstract model is intended to conceptually derive all possible states that the program may reach at run time. The soundness of the analysis then can be proved in a rigorous mathematic way. The algorithms of the analysis are used to simulate all the possible program behaviours and inputs arising dynamically at run time, rather than actually executing the program in a computer.

[1] The SLAM project originated in Microsoft Research in early 2000. Its goal was to automatically check that a C program correctly uses the interface to an external library.

1.2 Motivation

JavaScript has become the most widely used language for client-side web programming (RLBV10). The dynamic features and flexible characteristics of the language make understanding its program codes notoriously hard, and the lack of adequate static analysis tools make it hard to ensure the safety and correctness of its programs. Therefore, it is highly desirable to develop a framework to formally describe the behaviour of JavaScript programs and reasoning about them.

The design principles within JavaScript were taken from the *Self* and *Scheme* language and influenced by the C, Java, Python, and Perl languages. JavaScript follows certain conventions such as shared mutable data structures which means that one data structure could be pointed to or referred to by multiple pointers or references. These pointers or references are alias to each other, and the data structure could be modified after creation by any access path. This makes JavaScript programs even harder to keep track of the properties of its data structure statically than Java programs because of these dynamic features.

The emergence of separation logic (IO01; Rey02) promotes scalable and precise reasoning via explicit separation of structural properties over heap memory where recursive data structures are dynamically allocated. It enables the automated verification and analysis of heap manipulating programs. Being an extension of Hoare logic (Hoa69), separation logic is used to verify the functional correctness of programs by modelling *stack and heap* memory in a natural and accurate manner. A number of approaches have been made to automatically verify programs written in mainstream imperative languages such as C, C++, and Java (CDNQ08; BCO05; NDQC07; RL10; CDNQ10). Another advantage of using separation logic is to verify other properties of programs, rather than just memory safety and functional correctness. For example, the *shape property* [1]

[1] In Chin et al. (2007), shape property refers to the expected forms of some linked data structures, such as cyclic lists, doubly-linked list, height-balanced trees and sorted lists trees.

of linked lists and trees, the length of list and sorted order of list, the height of a tree and binary search property (CDNQ07; CDNQ08; CDNQ10; LHQ08; NDQC07). Using separation logic to verify the functional correctness of JavaScript programs has been an open research problem.

Another motivation of this thesis is to improve the quality of JavaScript client-side programs. As the programs are executed on the client platform, not only the correctness but also the safety of the programs are of concern to both the developers and end users. In this case, the client-side third party applications written in JavaScript may inject vulnerabilities into the host pages, which becomes a prominent threat. Comparing with the main stream programming languages, there is no adequate formal system for verifying the safety of third party JavaScript programs with respect to the host page. Therefore, the program would be more robust if a framework could also improve the safety of the program applications.

1.3 Objectives

The main objective of this thesis is to increase the level of automation of JavaScript program verification. The investigated aspects of the programs are functional correctness, and safety of third party application with respect to the host page. This thesis aims to develop an axiomatic system for verifying JavaScript programs. More specifically, the goal can be described in detail as follows:

Program Logic The first objective is to develop a program logic to automatically reason about a broad subset of JavaScript, including challenging features such as object field modification on the fly, function object, prototype inheritance, and scope chain. Logical reasoning has much to offer JavaScript, such as a formal description of program behaviour, and the ability to verify more general properties of the program, such as safety.

Unsafe program Discovery The second objective is to infer the precondition and postcondition of the program state, so that the generated specification can be precise enough for further analysis. For example, the specification can be progressed to analyse reachability relationships among objects. After further analysis, the objects that establish the reachability to the object *window* will have the ability to visit all the objects enclosed in the web page could be abstractly defined in a set. According to analysis of such a set, objects that directly or indirectly take on the authority of *window* can be discovered.

1.4 Criteria for Success

This section defines the measures of accomplishment to be used to evaluate this research, a series of criteria for assessing the success of the research shows as follows:

1. **Definition of a suitable and a safe subset of JavaScript**

   This research will define a suitable subset of JavaScript that can be used in the formal analysis for functional correctness and safety properties. An extremely small (trivial) subset would narrow down the range of its application functionality, whereas full JavaScript features are difficult to be verified using formal methods. Therefore, this criterion specifies whether the language has the trade-off between the expressiveness of the subset language and the feasibility with formal method verification.

   Generally, the safety property of a program refers to memory safety. In this thesis, a program written in the subset of JavaScript is defined to be safe if and only if the program does not maliciously interfere with the website host, especially the arbitrary manipulation of global variables on the host.

2. **Define Operational and Axiomatic semantics of the JavaScript subset**

For a verification framework, the underlying operational semantics must be presented to fully describe how a valid subset language program is interpreted as sequence of computational steps in a mathematically rigorous way. The underlying axiomatic semantics will be provided to prove the functional correctness of the subset language programs. The pre-condition and post-condition assertions written in logical statements will describe the meaning and status of statements of the program.

3. **Definition of safety verification algorithm**

   A verification framework must construct a specification language that is used to define predicates to specify program safety properties. The safety property must be formally defined in a rigorous mathematical way, and a verification algorithm should be designed to assist users to decide whether or not programs are safe.

4. **Proofs of program written in the JavaScript subset**

   As a verification system, the soundness property is the most fundamental property that needs to be proved. In this thesis, the system is sound if and only if its axiomatic inference rules prove only assertions that are valid with respect to its operational semantics.

1.5 Thesis Organization

This thesis is constructed using 6 chapters including current introduction Chapter 1. Figure 1.1 displays the interconnection relationships between the chapters. In Figure 1.1, the oval box represents the program verification topic, the diamond boxes represent properties of the language, and the rectangle boxes represent the reviews and approaches that have been employed.

Chapter 2 provides the state-of-the-art literature survey of the language features of JavaScript and program verification techniques, especially focusing on separation logic inference system, as well as the dynamic features of JavaScript.

Chapter 3 presents a subset of JavaScript, together with an axiomatic framework. The framework is composed of the operational semantics for strong soundness proof, the specification language describing the abstract domain, and the inference rules to automatically deduce the specification of the program states. The aim of this chapter is to develop a framework for verifying the correctness of the subset language.

Chapter 4 describes an upgraded axiomatic framework based on the one developed in Chapter 3. The language targeted in this chapter has more expressiveness than the one in Chapter 3, together with its safety issue. The system proposed in this chapter is able to verify both functional correctness and safety properties.

Chapter 5 presents the experimental results and evaluation of the axiomatic framework.

Chapter 6 concludes and summarises the contributions of the thesis and discusses possible directions for future work.

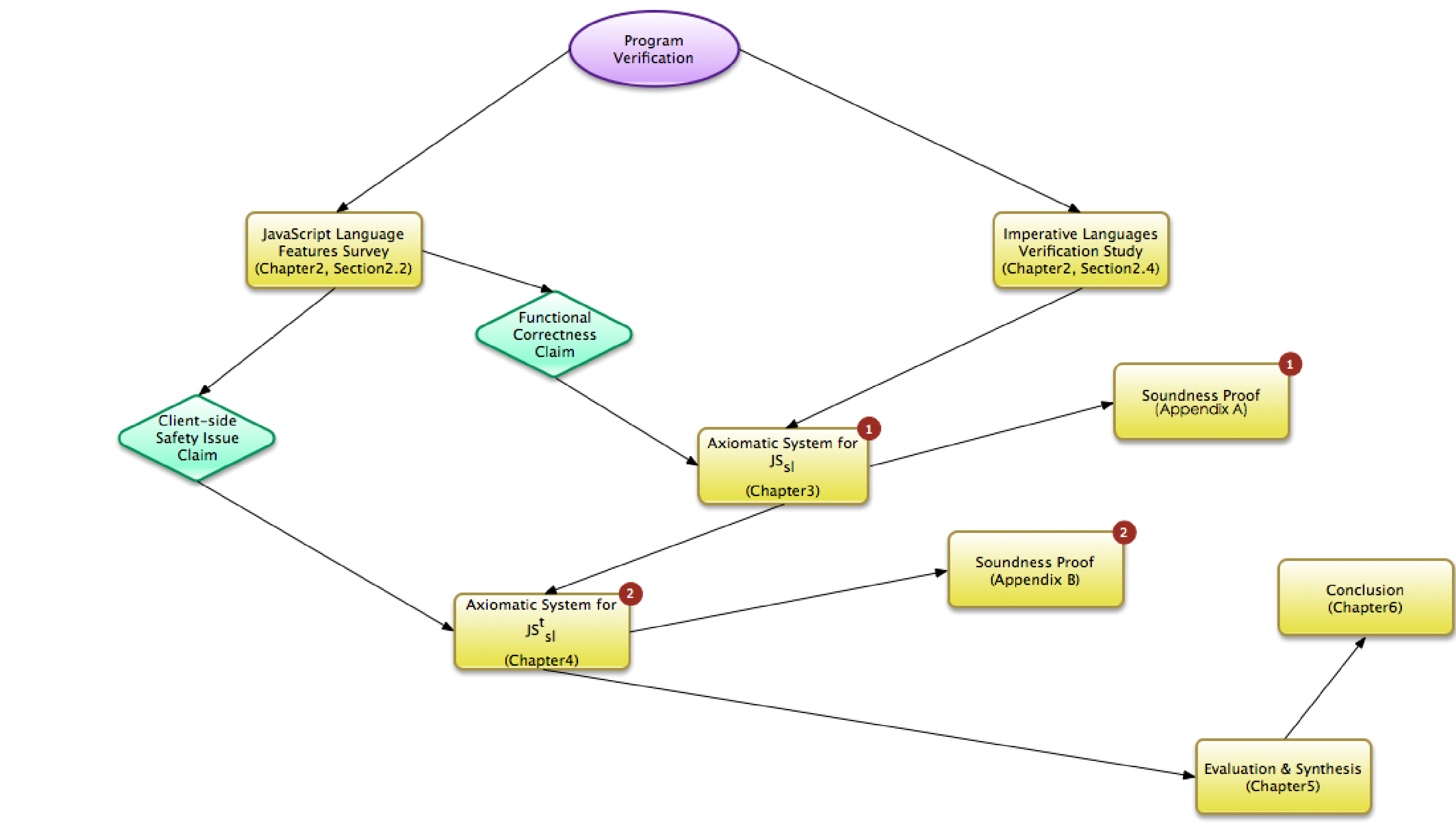

Figure 1.1: Thesis Structure

Chapter 2

Literature Review

2.1 Introduction

The discussion of the literature is divided into three parts. Firstly, the features of the JavaScript language that distinguish it from other programming languages are presented. A collection of examples are constructed to explain the features individually. Secondly, the state-of-the-art in the solutions for solving safety problem in JavaScript programs are introduced. JavaScript is adopted as a script language in a web environment, different web browsers use different JavaScript interpreters. Such diversity causes certain difference in performance. JavaScript has been widely used to be embedded in HTML document as third party applications to create dynamic performance. The HTML document needs to be protected from the safety issue caused by such applications. Thirdly, a brief survey of the related work on formal program analysis and verification is discussed. In particular, the techniques that might be suitable for verifying certain properties of the JavaScript language are presented.

2.2 JavaScript as a Scripting Language

In 1995, JavaScript was born for adding enhancements to the behaviour of web pages, primarily to web forms (Whi08). With the development of the Internet, the support for various web applications becomes extremely important. JavaScript was designed by Brendan Eich and developed by Netscape (Fla11). The desirable properties of JavaScript include that it was originally designed for increasing the interaction between the web users and web pages, communicate with the browser asynchronously. For example, online form applications, flash interactions. JavaScript has been implemented in most browsers, including Firefox, Safari, IE. Its major implementation are KJS, Rhino, SpiderMonkey, V8, WebKit, Carakan, Chakra. JavaScript was ranked the most popular programming language on GitHut and StackOverflow (Git12) in 2012.

JavaScript as a web scripting language has been widely used because of its remarkable expressive dynamic features. It is more productive and efficient to use JavaScript to construct applications on the client side with certain libraries (AJAX, Prototype, etc.) instead of using Java language. The comparisons between JavaScript and Java is employed in this thesis to answer the question of "what is JavaScript?". Table 2.1 shows general difference between JavaScript and Java from the aspects of execution, platform, web integration, etc. In comparison with Java, JavaScript has weak typing, prototyping is more concise, it is more suitable for building up small or middle-scale applications. This section is aim to provide an overview survey of JavaScript. The language features that are critical and distinct from other languages are chosen for discussion. The main outcome of this discussion is that JavaScript is powerful but flawed.

Item	JavaScript	Java
Compilation	It is not complied but runs interpretively by client.	It is compiled into bytecode (machine read-only) downloaded from server. JVM interprets the code to run on client.
Platform	It runs within browsers that support it.	It runs as applets within browsers that support it. It also can run as a standalone application on most platforms (Windows, Unix, Linux, etc.).
Web Ability	It can build dynamic webpages (forms, buttons, etc.) Its code embedded in HTML.	It can build Applet pages. For example, JSP (Java Server Pages) can build the webpages that contains HTML and JavaScript. Its applets distinct from HTML but can be accessed from HTML pages.
Web Presence	It develops applications efficiently. The downloading times of applications in webpages is reduced.	Its applets still exist on the web, but costs more time to perform.
Client Side	It does not allow direct access to a user's hard-drive. The source code can be viewed using "View Source", indirect source code also can be viewed by specified an external JavaScript file (using the SRC attribute).	It does not allow to access to memory or devices outside the applet except certain action that explicitly granted by permission. When its applet are compiled to bytecode and sent to the client side, the source code is not human-readable.
Programming Language	It is prototype-based. No distinction between types of objects. Inheritance is through the prototype mechanism, and fields and methods can be added to any object dynamically.	It is class-based. Objects are divided into classes and instances with all inheritance through the class hierarchy. Classes and instances cannot have fields or methods added dynamically.

Table 2.1: Overview Comparison Between JavaScript and Java

2.2.1 Main Features of JavaScript

There are three important concepts in JavaScript, *objects*, *functions* and *closures* (See Figure 2.1). The strong relationship among them builds up the foundations for any type of JavaScript application development. An *object* is an unordered collection of fields, each of which has a name and a value. Objects and fields can be modified at runtime. JavaScript treats functions as objects. A *function* can be created at runtime, stored in a data structure and returned as an argument for another function. In addition, JavaScript sustains *closure*, which permits functions intrinsically bound to variables outside their own scope, even when the scope is no longer visible.

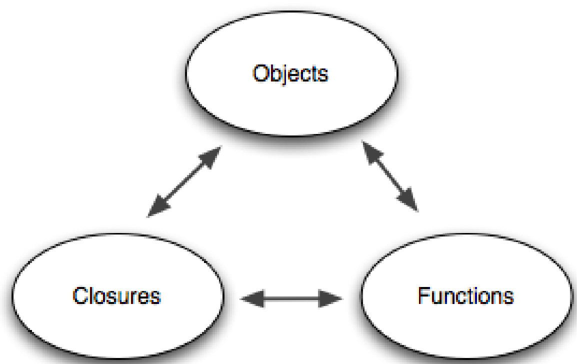

Figure 2.1: Three Important Concepts in JavaScript

2.2.1.1 Data Type and Variable

JavaScript contains a group of data types. The primitive types are *boolean*, *number*, *string*, *null* and *undefined*. The rest are of type object, a function is a type of object. Type-checking mainly happens at runtime. A variable is not declared with a specific data type, the type of a variable is determined by the latest value assigned to it. The language provides the *var* keyword to declare a variable which can be assigned or have its value modified at any point in the program. Figure 2.2 shows that variables in JavaScript have

loose typing and can be modified without any type casting. Unlike, variable declaration in Java that must be specified with data types and initial value, and a variable cannot change data type unless forced to by type casting.

2.2.1.2 Object Structure

In JavaScript, an object is an unordered collection of key value pairs. The keys of objects are fields. Fields are containers for primitive type values and other objects. This object structure can be specified in a variety of ways. In the case of object literals that is an object declared with a set of fields and correspondent values.

Figure 2.3 shows an example of how JavaScript declares an object x with fields a, b, and f in an object literal way. The field a has integer value 10, the filed b is a literal object that has field c, and the field f is a method of object x. In JavaScript language, fields are not explicitly defined, and can be added or removed at any time of program execution. Furthermore, JavaScript allows multiple fields access, for example, $x.b.c$ is actually two level access. Note that access to non-existent fields in JavaScript would not generate an error but return $undefined$ value. Attempting to access a field on the $undefined$ value will result in a runtime error.

2.2.1.3 Functions

A function is a block of JavaScript code which is defined once but might be invoked several times. A function block definition contains a function name (identifier), a list of optional parameters, local variables and body of the function. JavaScript functions have three important properties. First of all, functions are object, as such they have fields and methods. JavaScript supports first-class functions, which can be assigned to variables, passed as parameters. Note that any reference to a function allows it to be invoked using the () operator. Secondly, JavaScript supports nested functions. A function can define an inner function, the scope of the function includes local variables,

```
1  <script type = "text/javascript">
2
3  var a = 10,
4      b = "Hello World";
5
6  a = b;
7  document.write(a);        //"Hello World"
8  </script>
```

Figure 2.2: Variables Declaration in JavaScript

```
1  <script type="text/javascript">
2  var x = {
3          a : 10,
4          b : { c : 20 },
5          f : function () {
6                  return "Field f is a method.";
7          }
8  }
9  document.write(x.a);            //10
10 document.write(x.b.c);          //20
11 document.write(x.f());          //"Field f is a method."
12 <\script>
```

Figure 2.3: Objects Structure in JavaScript

inner function object and parameters. Thirdly, JavaScript supports closure. Since the creation of a nest function defines a lexical scope, this means that a function can be executed using the variable scope that was in effect when it was defined, not the variable scope that is in effect when it is invoked (Fla11). A closure is created when an inner function has access to the local variables located in the outer function. It is generated whenever the inner function is returned. For example, in Figure 2.4, the outer function f contains an inner function g. The return of the inner function generates a closure that is composed of the scope of function g and the local variable a. Note that, the variable a is declared outside of the inner function g. Technically, all JavaScript functions are closures and they have a scope chain associated with them. A function object is also created with a $Variable\ Object$ (VO) that refers to its current execution context. A $Variable\ Object$ can visit its local variables, function expression, and parameters that

are declared in its context. Variable identifiers are resolved against the scope chain. The identifier resolution starts with the first $Variable\ Object$ in the scope chain. It is checked to see whether it has a field with a name that corresponds with the requested identifier. The scope chain is a chain of $Variable\ Object$, if the request identifier can not be encompassed in the first $Variable\ Object$, the resolution traverses the scope chain to the next $Variable\ Object$ until the identifier is encountered. Note that multiple closures are allowed and are generated by the same nested function. The closures essentially save programmers from the need to create global variables all the time as they can keep a copy of the all the local variables.

In JavaScript, a function is called a method in the case of when a function is created as a field of an object. Objects can gain methods by assigning a function definition to a field. There are two ways to define a function in JavaScript (Ecm09), function declaration and function expression. A function declaration can be translated to a function expression (See Figure 2.5). The function declaration $function\ f()\{......\}$ can be converted into the function expression $var\ f = function()\{......\}$. In fact, the function name defined by a function expression can be used only inside the function body, whereas a function declaration creates a variable with the same name as the function name. Thus, a function declaration can be accessed by its name in the scope it was defined in. Another crucial difference is that function declarations are evaluated before the enclosing scope is executed, but function expressions are evaluated as they are encountered.

2.2.1.4 Prototype and Inheritance

In JavaScript, an object contains a sequence of fields and an internal field [[*prototype*]] [1] linking to its "super" object. When fields are accessed from an object by the dot notation, it returns the value if the field can be found in the current object, otherwise the field will be examined in "super" object by following [[*prototype*]]. The "super"

[1] This field is not allowed to be accessed except in certain browser environments, such as Firefox. Firefox provides variable *__proto__* to acquire the authorisation to access [[*prototype*]] property.

```
1  <script type="text/javascript">
2  var a = "Goodbye World";
3  var f = function () {
4          var a = "Hello World";
5          var g = function () {
6                  document.write(a);
7          };
8          return g();
9  };
10 f();     // "Hello World"
11 <\script>
```

Figure 2.4: Closure in JavaScript

```
1  \\ Function Declaration
2
3  <script type="text/javascript">
4  function f () {
5          var a = "Hello World";
6          return a
7  };
8  f();     //"Hello World"
9  <\script>
```

```
1  \\ Function Expression
2
3  <script type="text/javascript">
4  var f = function () {
5          var a  = "Hello World";
6          return a;
7  };
8  f();     //"Hello World"
9  <\script>
```

Figure 2.5: Function Declaration vs. Function Expression

object is the prototype of the current object. If the current object does not contain the field, the examination follows prototype chain which is linked objects, and return *undefined* when the chain of link objects is exhausted. JavaScript uses the prototype chain to provide inheritance, whereas, in Java, each class is allowed to have one direct superclass, and each superclass may have an unlimited number of subclasses. In Figure 2.6, the example of the JavaScript shows that the object x is the prototype of the object y, the object y has the authority to visit the fields that are contained in the object x,

such as the field a and f.

```
1  <script type="text/javascript">
2  var x = {
3          a : 10,
4          f : function () { return "Hello World";}
5  };
6  var y = { a : 20 }
7
8  y.__proto__ = x;
9
10 y.a;     //20
11 y.f();   //"Hello World"
12
13 <\script>
```

Figure 2.6: Inheritance in JavaScript

In addition, JavaScript makes no distinction between constructors and other functions. Every function gets a *prototype* property that is used to store all the fields that the function's instance can inherit from. Figure 2.7 shows that a field y can be added to function object f by *.prototype* notation on the fly. This statement provides any instance of object f an additional field y. The notion of *prototype* and [[*prototype*]] is not well defined. Table 2.2 shows the difference between them.

```
1  <script type="text/javascript">
2  var f = function (x) {
3          this.x = x;
4  }
5  var g = new f("Hello");
6  g.x;     //"Hello"
7  f.prototype.y = 10;
8  g.y;     //10
9  <\script>
```

Figure 2.7: Inheritance of Function Object in JavaScript

Moreover, *prototype* is one of the function object properties that can be applied for fields inheritance, whereas [[*prototype*]] is an internal and hidden property pointed to an actual prototype (as superclass in Java). Note that, in Table 2.2, $getPrototypeOf()$ is an object property to check whether the object owns a field, *constructor* is set to

the function's *prototype* property at function creation. *Object.prototype* is the actual prototype of the object *Function.prototype*, and they can be distinguished by their dissimilar internal properties.

Context		Interpreter Browser	
		Firefox	IE
var f = {};	f.prototype;	undefined	undefined
	f.__proto__;	[object Object]	undefined
	f.constructor.prototype;	[object Object]	[object Object]
	Object.getPrototypeOf(f);	[object Object]	abort
	f.__proto__ ===Object.prototype;	True	False
`function f() {}` **OR** `var f=function(){}`	f.prototype;	[object Object]	[object Object]
	f.__proto__;	function () { }	undefined
	f.constructor.prototype;	function () { }	function prototype() { [native code] }
	f.prototype.constructor;	function f(){}	function f(){}
	Object.getPrototypeOf(f);	function () { }	abort
	f.__proto__ ===Object.prototype;	False	False
	f.prototype == f.__proto__;	False	False
	f.__proto__=== Function.prototype;	True	False
	Function.prototype.__proto__ === Object.prototype	True	False
	f.__proto__== Function.prototype;	True	False
`function f() {}` `var g = new f();`	g.__proto__ === f.prototype	True	False
	g.constructor === f	True	True
	g.constructor.prototype == f	False	False
	g.constructor.prototype === f.prototype	True	True

Table 2.2: Dot prototype vs. Dot [[prototype]]

2.2.1.5 Other Conventions

There are some other features of JavaScript that need explanation.

- Execution Environment

  JavaScript codes rely on an execution environment to be embedded in or to be included in HTML pages, which is able to perform interactions with the Document Object Model (DOM) of the page (Ecm09). The code should be either stored in public (the HTML document) or delivered as a *.js* external file, and they are interpreted by an interpreter in the browser on the client-side. In addition, the interpreters treat unassigned variables, unassigned object fields, functions that do not have return statement as *undefined*. The value *null* is a type of object that represents object value and the interpreters return *null* when a variable is set to an empty value.

- Pass By Value or Reference

  In JavaScript, functions parameters are passed either by value or reference. It depends on the type of the parameter, passing by value if they are primitive type and passing by reference otherwise. In the case of passing by value, the action has no effect on the original variables, whereas if the parameters are passed by reference, the action on them actually copies in the records referred by the reference and also copies out when the operation of the function is finished. It means that the modification of such kind parameters changes their records referenced (values) by the locations.

- Metaprogramming

  JavaScript allows code to be represented as strings and executed. For example, Figure 2.8 shows that a sequence of code is assigned to variable *prog* as type *string*, the execution of *eval* function cause the execution of the sequence of code. The *eval* function is used to execute JavaScript expression, statement, or sequence

of statements. The variables declared in a *eval* function is considered as global variables. The codes passed to the *eval* is executed with the privileges of the interpreters. Thus malicious codes can be executed within the code that passed to the *eval*.

```
1 <script type="text/javascript">
2 var prog = "x=10;y=20;document.write(x*y)";
3 
4 eval(prog);       //200
5 <\script>
```

Figure 2.8: Metaprogramming in JavaScript

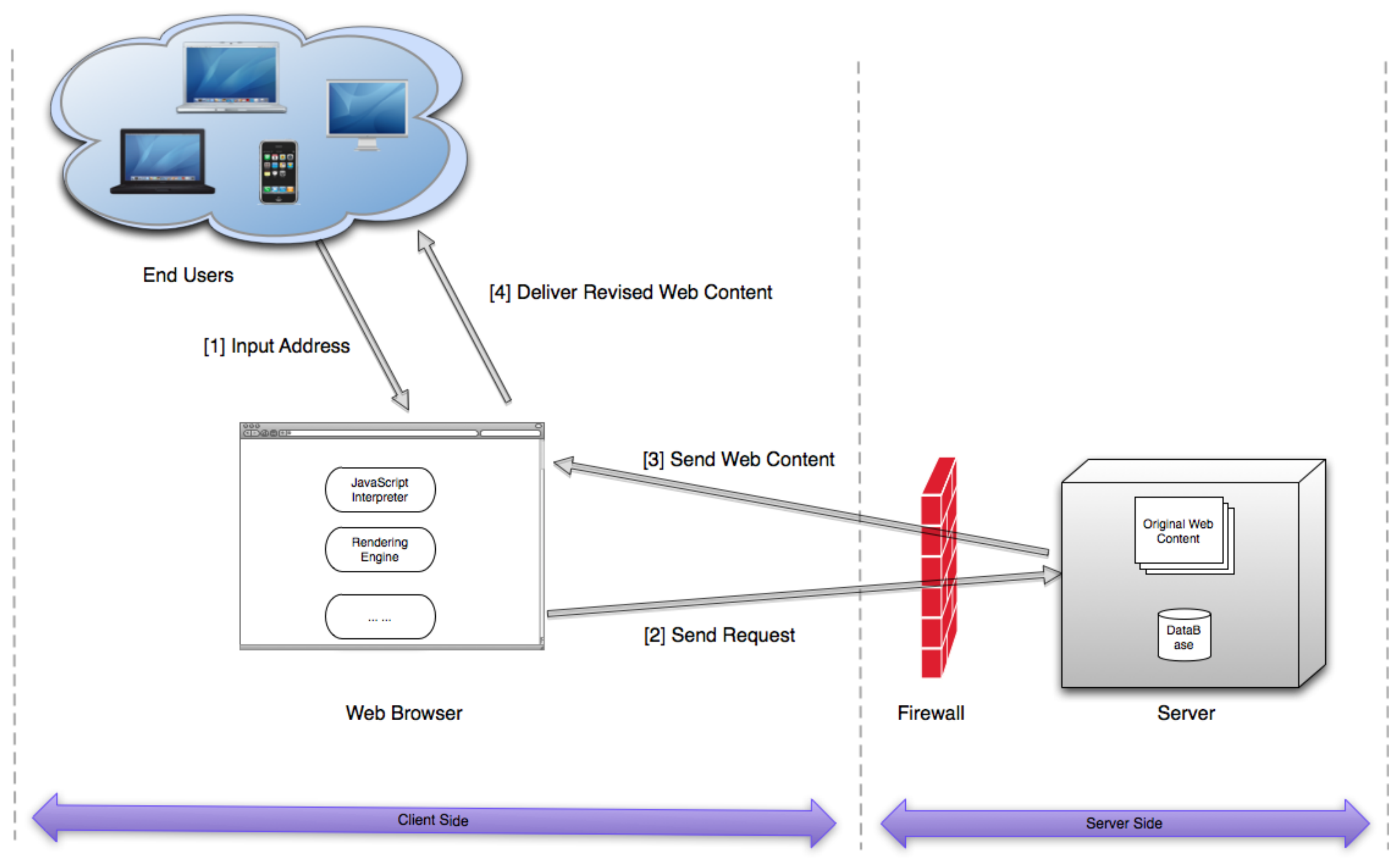

Figure 2.9: Overview of Web Browser Implementation

2.2.2 Client-Side JavaScript

JavaScript mainly allows programmers to write programs to perform computations in the client side (the user's Web browser). JavaScript is not originally designed for interacting with a database on the serve. In Figure 2.9, the process of how a Web browser retrieves, presents, and translates contents on web pages is shown. In the *step one* ([1]), the end users give valid web address in the context of the web browser. The *step two* ([2]) process sends the request to the *server* through the firewall. Note that, the server owns the original web content and the database. After that, the Web browser downloads the requested web content back from the *server* in *step three* ([3]). Then the browser scans the web content (document), and delivers the revised web content to the end user which has been "translated" into human readable form in *step four* ([4]). Inside the Web browser, internal engines such as a rendering engine and a JavaScript interpreter are responsible for the "translation" process.

Each browser has its individual JavaScript interpreter. The first JavaScript interpreter SpiderMonkey, was created at netscape for the Navigator Web browser (KK97) and implemented using the C language. Mozilla constructed the Rhino engine which was developed in Java (Rhiva) for interpreting JavaScript programs. The existing interpreters include TraceMonkey in Firefox (Trane), V8 in Chrome (V81ne), Carakan in Opera (Carom), JavaScriptCore (also named Nitro) in Safari (Nitit). As these interpreters were developed under different semantic model definition, they may return different output when executing the same piece of of JavaScript program.

The primary objective of a web browser is to display HTML documents in a window. The execution environment of the HTML document is considered as a global execution context. In the context of the client-side JavaScript, the *window* object represents the reference of the global object in the global execution context with respect to client-side programming (Ecm09). It has the authority of manipulating arbitrary properties and methods in the context of the HTML document (DomOM). Therefore the permission

to access the *window* reflects the permission to manipulate all the global variables.

In addition, JavaScript also has an influence on third party application development in a web page (host page). A web page may host more than one piece [1] of JavaScript code, any such piece could be used to publish advertisements or widgets [2] that are maintained by third parties. Therefore, a malicious third party who has the authority to access the *window* object is able to compromise the entire host page. In such a case, the piece of third party code (guest code) is not safe with respect to the host page. To solve this safety issue, several mainstream approaches have been introduced, including *developing new languages*, *restriction rewriting and wrapper − based isolation*, *sandbox virtual machine*, and *statically verified containment*. They are explained as follows:

Developing New Languages JavaScript language is in full of problems, including unsafe and insecure. An intuitive solution is developing a brand new language that performs the same functionality and more robust than JavaScript. Google published a *DART* language aiming to replace JavaScript on the Web ultimately. The interesting features it supports are optional static typing and single inheritance (Darge). Essentially, *DART* is intended to solve the fundamental flaws of JavaScript including memory leak and safety breaching, whilst providing better performance.

The problem of developing a new mature language would cause hugh barrier for adoption.

Restriction Rewriting and Wrapper-based Isolation This approach defines safe subsets of JavaScript which blacklists some features of the language that are considered as dangerous, rewrites the rest of the features, and wraps the guest code

[1] A block of JavaScript starts from the tag <script type="text/javascript"> and end at the tag </script>.

[2] A widget is a programmed HTML fragment with some script. That script can manipulate the HTML fragment but not be able to manipulate any other part of the document.

into an enclosed function.

Caja is a project developed by Google (Cajer). It enforces the untrusted guest code to be executed by a runtime check framework. Caja uses an *object−capability security* model to allow a wide range of flexible polices that are intended to make the guest code safer with respect to the host page. In this model, the objects can only be changed through the references they hold, objects can only receive references through methods calls, objects never start with references, the use of functions as objects is forbidden and encapsulation is enforced. This model encapsulates the JavaScript codes into a single JavaScript function with no free variables. However the problems with Caja are that it does not guarantee any particular safe fields and the rewriting process may cause massive workloads and the runtime cost.

Another example of using rewriting approach is that of Facebook which defines the FBJS language to develop integrated JavaScript applications on the Facebook web page. FBJS is subset of HTML and JavaScript (Fbjpt). Essentially, it removes certain safety-critical constructs (such as *eval*, *Function*, *constructor*), rewrite others (such as *this*) to permit them to be used safely, and enforce various wrapped DOM functions to provide control to the DOM objects. The FBJS library is applied to attach namespaces for objects in the guest code. However, a safety issue was found by Maffeis et al. (MMT09b; MMT09a) in 2009. The issue shows that an object with no capability to modify native objects can indirectly gain such capability without being granted by the host page. For example, an assignment of the reference *Obj.toString* to a new object which *Obj* was defined in advance. The new object obviously can manipulate the function *toString*() that is one of the methods of the native object *Object.prototype*. There are still more vulnerability issues that need to repaired in FBJS (MMT10).

The BrowserShield (RDWD07) system was built to perform dynamic instrumenta-

tion of embedded scripts and employ the vulnerability-driven filtering policies for customised runtime activities. It rewrites the web pages and embeds JavaScripts into safe equivalents that contain logic for recursively applying runtime checks to keep modifying content based on known vulnerabilities. However, this system suffers from massive runtime overhead, and a systematic way of guaranteeing the correctness of the enforced policies has not been provided.

Guarnieri and Livshits (GL09) stated that the GateKeeper system provides a mostly static approach for enforcing safety and reliability policies for JavaScript programs. However, the language they focused on is less expressive than others work (MMT10; Fbjpt; Cajer; RDWD07).

In fact, the $restriction\ rewriting\ and\ wrapper-based\ Isolation$ approach may cause incompatibility problem and increase the difficulties of the connection between two different isolation systems that are built on the different versions of JavaScript.

Sandbox Virtual Machine In comparison with Java, JavaScript does not have any built-in sandbox mechanism [1]. ADSandbox (DHF10) was introduced to execute embedded JavaScript within an sandbox environment and record a log of critical actions. Before the web content is displayed in the browser, the browser invokes a $Browser\ Helper\ Object$ that can hand over the website URL to an $Dynamic\ Link\ Library$ (DLL). This DLL downloads the web content from the requested web page in order to analyse it. The analysed result is recorded in a log, and the users would be navigated to an error page if any suspect action is detected. The sandbox virtual machine provides a technically sound approach to permit backward compatibility with current APIs.

Statically Verified Containment To protect the host pages from malicious third

[1] A sandbox mechanism provides a set of resources for unverified third party programs to run in. It is used to executed untested codes.

party applications, one promising approach is statically verifying the JavaScript programs in terms of the properties that keep the host pages safe. ADsafe (Crong), Dojo Secure (Dojit), and Jacaranda (Jacty) are existing JavaScript verifiers. The verification is based on a subset of the target language. The main idea of this approach is blacklisting known dangerous fields or properties that would breach the given policies, including accessing global variables (such as *document*), dangerous internal properties (such as @*constructor*), and allowing unverifiable constructs (such as *eval*). For example, ADsafe removes the features that cause capability leakage, it blocks the guest code from directly or indirectly accessing any global variables, such as the variable *this*. The variable *this* is bound to the global object *window* when a method is invoked as a function. It can implicitly access the object *window*. More details about ADsafe are discussed in Section 2.2.3. Note that only untrusted guest code needs to conform to the subset ADsafe, the code from the trusted third party has access to the entire JavaScript language. This approach has no measurable runtime cost and no extra rewrite workloads, and it is compatible with independent tools.

The limitation of this approach is that it can only protect host page from guest code, but cannot protect guest code from host page. Meanwhile, the expressiveness of the language is reduced when certain features of the language is removed. Furthermore, the renting portion of the host page also could be resold to another advertisement network, which makes host page even harder to protect. In fact, the existing safe subset containments still show that they fall short of defending against capability leaks. In the case of adding a method to the built-in prototypes for a certain purpose in host page, the malicious guest code which observes this method can use its capability to breach the containment and compromise the host page.

Finifter and Barth (MWB10) provided an improved solution. They whitelist known

safe fields using namespaces to prevent capability breach. Their solution instruments the browsers to load guest code and host page in separate heaps. The variables on these heaps are marked as vetted. The breach of the guest code can be detected when the instrumented browsers detect a suspicious reachability edge from the different heaps. The $Blancura\ Verifier$, an extension of ADsafe containment, whitelists all the loaded variables with two different namespaces attached for identifying which heap a variable is from. Thus, none of the functions created previously by the built-in objects from the host page are accessible by the guest code. The advantage of using this approach is no extra rewriting workloads. Adding prefix namespace to the variables does not measurably affect the runtime performance. However, it is difficult to exploit all suspicious reachability edges, and not all suspicious edges are intended to breach the containment. The subset language is even more restrict than ADsafe.

2.2.3 ADsafe

ADsafe defines a subset of JavaScript which permits host page to publish guest codes and perform valuable interactions, such as third party scripted advertising or widgets. Meanwhile, it can prevent malicious attacks or intrusion from guest codes. JSLint (Jslnt) provides the ADsafe subset language program a validation mechanism (only syntax checking) to release the inspection burden off a human's shoulder and review guest codes for safety checking. ADsafe relies on static analysis. It does no runtime checking or code rewriting, and it requires no program transformations. Essentially, it adds capability discipline by removing features that cause capability leakage, including that limited access to primitive type variable except $null$ and $undefined$ as those variables are global variables or can be linked to them. Another deleted feature is the use of **this** as when a method is invoked as a function, **this** is bound to the global object $window$. For example:

$var\ a = \{b : function()\{return\ this;\ \}\};$

$var\ test = a.b;$

$test();\ //window$

In the above example, **this** is an internal variable in every function object. When the method b is called as a function such as $a.b()$ in the global environment, the value of **this** is bound to the global object $window$. Another more complicated example is shown below:

$var\ a = \{b : function()\{$

$\quad var\ c = function()\{return\ this;\ \}$

$\quad return\ c();$

$\}\};$

$alert(a.b());\ //window$

This example presents the function invocation $a.b()$ in this program still returns **this** to $window$. As you can see, both example explain the potential dangerous of using variable **this**. Other main prohibited features in ADsafe are shown as follows:

- *eval* - The *eval* function can access the global object.
- *with* - The *with* statement can modifies the scope chain which increases the difficulty of static analysis.
- Dangerous Properties - Words which cannot be used are $apply$, $arguments$, $prototype$, $callee$, $caller$, $constructor$, $stack$, $unwatch$, $valueOf$, and $watch$. Because the use of those variable names are able to cause capability leakage in some browsers.

- Words starting with "_" cannot be used. Because some browsers recognise dangerous properties or methods having a trailing "_".

```
1  <html>
2  <head>
3  </head>
4  <body>
5
6
7  <script src="http://www.yourTrustedDomain.../adsafe.js"></script>
8  //load minimized adsafe.js library
9
10
11
12
13 <div id="TEMPLATE_">
14 <script>ADSAFE.id("WIDGET_NAME");</script>
15
16
17
18 <script src="ALIB.js"></script>
19 // load additional library
20
21 // ADSAFE.lib("ALIB", function () {
22 "use strict";
23
24              //return{
25              //additional functions
26              //}
27
28              //  });
29
30 <script>
31 ADSAFE.go("TEMPLATE_", function (dom, lib) {
32 "use strict";
33
34 //Codes Here
35 //can call the function from ALIB.js at lib.ALIB.funName()
36
37
38 });
39 </script>
40
41
42 </div>
43 </body>
44 </html>
```

Figure 2.10: ADsafe Structure in HTML Document

Furthermore, the ADsafe subset prevents guest codes from accessing the global variables or the *Document Object Model* (DOM) elements directly. This mechanism simplifies the safety control for the host page with respect to the guest codes by enforcing the scripts to access an *ADSAFE* object which is provided by host page, and offering indirect access to the guest code's DOM elements. Figure 2.10 shows that a widget is

generated in a HTML document with the name "$TEMPLATE_$", and the codes for the widget is wrapped in the function $ADSAFE.go(..., function(dom, lib))\{...\}$. The guest codes in this block can access the document through the *dom* parameter, permitting it indirectly access to HTML elements and allowing it to modify content, styling and behaviour. The parameter *lib* allows it to access the library file. The library file $adsafe.js$ is a AJAX library that provides the widget with limit access to DOM.

In fact, ADsafe does not provide a mechanism to guarantee the program to be safe. A malicious third party can redefine the existing $adsafe.js$ library to breach the safety feature of the sandbox. ADsafe can only protect host page from guest codes, but it cannot protect guest codes from host page. This is why we need a logic proof system to provide a more precise and accurate framework.

2.2.4 Summary

The flexibility of the JavaScript language was presented by the explanation of its main features. The issues caused by these features contain memory leak, capability leak, and safety beach:

- $Memory\ leak$. A memory leak issue happens when an object of the JavaScript program that is stored in memory cannot release corresponding memory after the execution is finished.

- *Capability Leak*. When a variable from guest code is able to perform the capability of built-in objects that are located in host pages, meanwhile the host pages have the new properties of these built-in object created in advance, this scenario may cause a capability leak.

- $Safety\ Breach$. An untrustworthy third party JavaScript application may breach the protections on host pages by using a malicious operation. For example, a third party application that maliciously manipulates the *this* variable can gain access to

the *window* object, which has the ability to access all the global variables. Even of the third parties' trust is established, the malicious syndicated third parties also can breach the trust establishment and cause indirectly damage to the host page. The approach to prevent such damage is building up a verification framework for detecting the malicious behaviour and improving the safety of the host page.

2.3 Formal Verification

Formal verification provides a foundation for the techniques that are used to build a mathematically rigorous model for a complex system (Har02). A complex system requires more accurate analysis and quality guarantee. It is possible to improve reliability of a system by verifying certain properties of the system. Formal verification has been proposed to reason about the quality of softwares since 1960s (Hoa69; Flo67).

Perlis and Backus (PS58; BBG^{+}60) introduced the ALGOL that was proposed to be an universal language for describing programming languages by notations. McCarthy (McC63) applied ALGOL to introduce meaning of programming languages based on evaluation of recursive functions. Floyd (Flo67) proposed the way of adding assertions to flowcharts of programs rather than by using ALGOL, in such way that a rigorous standard is established for proofs about programs, such as proofs of functional correctness. The assertions he developed can be assigned conditions at each branch and entry point in the flowcharts of programs. The conditions referring to the value of variables ensure that if these conditions were true upon entry, thus they can be proven true at exit. The proof of correctness of programs is rested on the proof that a program satisfies its specification. Hoare (Hoa69) made two additional steps upon Floyds work. Firstly, he discarded the flowcharts and developed a axiomatic system for reasoning about programs using specifications of statement behaviour that have become known as Hoare Triple, which is composed of three parts, P C Q, the precondition P, the statement C,

and the postcondition Q. Secondly, he argued that the axiomatic system could be viewed as an abstract foundation of recording the semantics of programming languages. This has the profound effect of opening up a way of developing provable programs rather than viewing their verification as a post hoc concern.

The process of software verification consists of numerous techniques and tools, often used in combination with one another (Col98). Program verification refers to the process of determining whether or not the products of a given phase of a software development process fit the requirements or purpose that established during the previous phase (IEE83). In fact, abstract data types, operational semantics, axiomatic semantics, specification languages all draw on formal verification(How87). There are two main aspects in formal verification, verification techniques and formal frameworks. A verification technique is applied to reason about desired property in a program with a corresponding implementation. The possible techniques are model checking, automata theoretic techniques, automated theorem proving (CM99). Formal frameworks (CM99) contain specification language that is used to describe the desired property of a program. The language can be developed under temporal logic (BR01; Eme81; Pnu77), predicate logic (BMMR01; CDNQ10), Hoare logic (Flo67; Hoa69), and Separation logic (GMS12). A formal framework also consists of a set of axioms and inference rules with theorems to be proved. The existing formal frameworks include interactive theorem prover *Coq* (Coqnt), the proof assistant *Isabelle* (Isant), the mechanical theorem prover $ALC2$ (Alc50), and a separation logic based verification framework Smallfoot (BCO05). These frameworks allow the expression of mathematical assertions and mechanically checks proofs of these assertion. Proof of program correctness is a technique attempting to identify program's faults or errors that cause failures (Col98). Floyd and Hoare had two foundational papers for program verification (Flo67; Hoa69). They introduced the concept of proof of correctness that composed of partial and total correctness and built up the logical base of program verification.

2.3.1 Model Checking

The notion of performing software verification with logic model checking techniques has evolved from intellectual curiosity to applicable technology (HJG08). Model checking has been researched for a number of years (CGP99) and achieved great success in circuit design and implementation in 1992 (McM92). It was originally introduced by Clarke and Emerson (CE81), later by Sifakis (QS82). It was designed to verify finite-state systems by exhausting the entire set of computation states according to some specification described in temporal logic. Essentially, the verification is converted to a formulae for satisfaction checking, $\mu \models \phi$, where μ is a finite model in an appropriate logic for representing the system, the specification is represented by the formula ϕ and the verification method refers to compute whether or not the model μ satisfies ϕ ($\mu \models \phi$) (HR04). The checking procedure is completely automatic and fast, and it will either terminate with answer true or give a counterexample execution to show why the model does not fulfils its intended purpose. A true answer means the model satisfies the specification (CGP99). The techniques for applying model checking are temporal logic (Pnu77), abstraction (BMMR01) and counter example refinement (CGJ$^+$00; CGJL03). Temporal logic was proposed for reasoning in computer programs (Bur74; Kro77; Pnu77). Pnuili (Pnu77) was first to use temporal logic for reasoning about program properties from a set of axioms that described the behaviour of the statements in that program. The introduction of a temporal logic model checking algorithm was original in the early 1980s by Clarke (CGP99) and Emerson (Eme81).

A model checker proposed by Braghin et al. (BSB07), was introduced to formalize and automatically verify the mobile system programmed in JavaScript. Their model checking engine takes the abstract programs that are pre-processed by the original programs and a set of security policies as an input, and provides a configuration document that describes whether or not the current input violates the given policies as an output. This work enables the specification of generic security policies for JavaScript programs

to make access control and information flow policies possible to be defined.

2.3.2 Hoare Logic and Verification

Hoare logic sets up a logic foundation framework for program verification (Flo67; Hoa69). It uses logical formulae to depict the behaviours of the programs, and introduces an axiomatic method that consists of a set of axioms and inference rules for rigorously deductive reasoning the static functional correctness proofs of programs mathematically. The essence of Hoare logic is the Hoare triple $\{P\}C\{Q\}$. Both P and Q are logic assertions at specific program point. P is a precondition, Q is a postcondition and C stands for the program statement, such as assignment, function invocation and so on. Moreover, Hoare logic used Floyd's (Hoa69) to define axiomatic semantics for programming language as a proof system. The axiomatic semantics mathematically define the semantic of a statement in a program by describing its effect on assertions. When the precondition P is met, the statement C establishes the postcondition Q. The correctness of a program reduced to reason about individual statements. Hoare logic only support partial correctness which means that the functional correctness of programs can be proved not including program termination. If C is executed in an assertion initially satisfying P and it terminates, then the final assertion satisfies Q.

A later work (Bur74) integrates operational semantics into Hoare logic for assisting on the soundness proof of a formal framework . Manna and Pnueli (Hoa69; tTCoP) stated that soundness of a formal framework is a fundamental property expected to be proved. Slonneger and Kurtz (SK95) declared that formal framework constructed as an axiomatic system is sound under *given semantics* with respect to *underlying semantics*. The operational semantics provide an *underlying semantics* in the form of a set of operational semantic rules to describe how a valid program is interpreted as a sequence of computational steps. The operational semantics is classified into two categories, *structure operational semantics* (small-step semantics), which formally describe how

the individual steps of a computation take place, and *natural semantics* (big-step semantics), which describes how the overall results of the executions are obtained. Moreover, *underlying semantics* refers to a concrete model which indicates the validation of the language that is to be verified. The language can be a core of Java, C or JavaScript. Plotkin (Plo81; Plo04) firstly introduced Plotkin style *structural operational semantics* (SOS) for defining the behaviour of a program in a structural approach. This approach contains a significant amount of information details that describes the semantics of a program in terms of transition relation in the way of a higher level understanding the program. A set of transition rules define the valid statement transitions. For example, $< x, \{x \mapsto 3, y \mapsto 2\} > \longrightarrow < 3, \{x \mapsto 3, y \mapsto 2\} >$ represents the application on variable evaluation rule. It shows that the evaluation of the variable x in the program state $\{x \mapsto 3, y \mapsto 2\}$ will update or remain the state with the assignment $x = 3$. A number of research (Win93; Rey98; Ros98; Sch00) showed that SOS has become a popular method for describing language semantics in program verification. Colvin and Hayes (CH11) proposed a different method which the transition arrows are labelled by the behaviour associated with the corresponding step. For example, $x \xrightarrow{x=2} 2$. Their labelled style operational semantics are able to describe the program semantics more abstractly and intuitively especially for concurrent programs. Middelkoop et al. (MHK04) employed a separation logic style operational semantic to deal with the feature of dynamic function invocations and field updates in Object Oriented Languages. Their method more accurately describes the status of stack and heap in the transactions. For example, $< x = 2, (s, h) > \rightsquigarrow ((s \mid x \to 2), h)$ shows that the execution of the assignment $x = 2$ extends the stack with $x \to 2$ after the transition. The objective of operational semantics is to present the correctness of the implementation of the language in a formal framework.

The correctness proof of program relies on *given semantics* in a formal framework that refers to an abstract model which uses assertions to describe the meaning of a

statement in a specification language. A specification language is generated to describe the program at a higher level than a programming language. It can be constructed on algebraic structures and Hoare logic, which consists of a collection of sets of data types and statements. This abstract model takes the validation level to verification level which is able to prove program properties formally.

2.3.3 Separation Logic and Verification

For modelling the complexity of program memory state, separation logic can strengthen the applicability and scalability of program verification for imperative programs by using shared mutable data structures (Rey00; Rey02; Rey05). Separation logic is an extension of Hoare logic, it remains the use of Hoare tripes to describe individual program states. Hoare triples asserts that if the program C executes from an initial state satisfying the precondition P, then the program will not go wrong and if it terminates, then the final state will satisfy the postcondition Q. Note that C might only access the memory locations whose existence is asserted in the precondition or that have been allocated by C itself.

In addition to the standard rules from Hoare logic, separation logic supports separation conjunction $*$ and spacial implication $-*$. The formula $\Delta_1 * \Delta_2$ asserts that two heaps described by Δ_1 and Δ_2 are domain-disjoint, while $\Delta_1 -* \Delta_2$ asserts that if the current heap is extended with a disjoint heap described by Δ_1, then Δ_2 holds in the extended heap that describes the final state. Such connectives are supported by a low-level storage model based on both stack and heap memory. In this model, four sets are assumed: **Loc** of memory locations, **Val** of primitive values (with $0 \in$ **Val** denoting *null*), **Var** of variables (program and logical variables), and **ObjVal** of object values stored in the heap, with $f_1 \mapsto v_1, ..., f_n \mapsto v_n$ denoting an object who has fields $f_1, ..., f_n$ with values $v_1, ..., v_n$. Then a concrete memory state h, s, consisting of heap and stack,

is from the following concrete domains:

$$h \in \mathbf{Heap} = \mathbf{Loc} \rightarrow_{fin} \mathbf{ObjVal}$$

$$s \in \mathbf{Stack} = \mathbf{Var} \rightarrow \mathbf{Val} \cup \mathbf{Loc}$$

Separation logic based model supports the basic program operations such as lookup, update, allocation and deallocation with a series of Hoare logic style reasoning rules. The $frame\ rule$ enables local reasoning possible in a formal framework. This allows the reasoning only follows the footprint of the program.

Figure 2.11 shows the base of local reasoning. Two domain disjointed heap P and R, Q and R. When the execution of the program statement C does not manipulate the state on the heap R, which means that the set of program variables modified by C do not share elements with the program variables describes in the state R, then the program verification refers to $\{P{*}R\}C\{Q{*}R\}$ can be concentrated on program footprint (the heap that the program actually manipulates) verification $\{P\}C\{Q\}$. This rule significantly reduces the scalability of program verification in terms of program state description.

Smallfoot (BCO05) is the first separation logic based formal framework. It is designed for checking the assertions of sequential and concurrent programs that manipulates dynamically allocated data structures. However, this framework concentrates on the verification of programs written in one specification language. Tuerk (Tue09) proposed a formal framework inside Higher-order logic [1] theorem prover that expresses different flavours of separation logic and makes it instantial for different programming language. The implementation of this framework is similar to Smallfoot. A $HIP/SLEEK$ (NDQC07) formal framework was developed to automatically verify the functional cor-

[1]Higher-order logic refers to quantification over properties and predicates rather than just object. For example, a higher-order sentence $\forall x\ \forall P(x \in P \lor x \notin P)$ denotes that for every individual x and every set P of individuals, either x is or is not an element of the set P. The set P is the property and x is an object.

$$\frac{\{P\}\ C\ \{Q\} \quad \mathit{modified}(C) \cap \mathit{vars}(R) = \emptyset}{\{P * R\}\ C\ \{Q * R\}}$$

where $\mathit{modified}(C)$ denotes the program variables modified by C, and $\mathit{vars}(R)$ represents the set of free variables in R.

Figure 2.11: Frame Rule in Separation Logic

rectness of heap manipulating programmes. HIP is a separation logic based verification system for a imperative language and able to verify the assertions of individual program state written in a specification language. $SLEEK$ is a fully automatic prover for taking a set of separation logic proof in the form of formula implications as input and decides the functional correctness of a program with a set of axioms and inference rules. A program logic based on separation logic for JavaScript has been presented by Gardner et al. (GMS12). Separation logic provides their work a way of modelling challenging features of JavaScript such as prototype inheritance.

2.3.4 JavaScript Program with Formal Framework

The dynamic features of JavaScript make the JavaScript programs a challenge to be verified. The third party applications of JavaScript on the Web indicates that there are two essential properties of the program required to be verified: functional correctness and safety of host codes. The definition of the functional correctness property is about a program operating correctly in response to its input/output. For each valid input, it produces the correct output. The definition of safety of host codes, is that the ability of third party applications mash-up to operate without causing failure regarding to web pages dependability. Moreover, a third party application (guest codes) cannot maliciously interfere with the host page (host codes) where it is located.

Recent research (Cro08; Crong; Fla11) shows that JavaScript web applications can run improperly in some circumstances and web pages suffer from attacks for years. The

vulnerabilities of a web browser (CMS+07; RDWD07) or a runtime attack (HYH+04; KKKJ) may directly harm a host page or registered users through certain malicious attacks, such as drive-by download (MBGL06), cross-site scripting (FGH+07), web privacy attack (BBN07).

A number of verification techniques and formal framework were developed to verify JavaScript programs. Yue et al. (YW09) declares the demand of safety verification in JavaScript applications. They presented a measurement study on unsafe JavaScript applications on the Web and provided an analysis results to indicate that it is necessary to ensure safety of these applications.

Temporal logic technique that model checking used for automatic verification hardly model the dynamic features of JavaScript (BSB07; HJG08) and state explosion problem (McM92) makes model checking suffer to scale to large JavaScript programs verification. A formal framework (AB04) based on Hoare logic was proposed to model safety property in a model checking system (TA05). They used information flow technique (GM82) and categorised the variables in JavaScript programs into high security and low security variables to ensure that low security variables do not flow to the critical operations that involves high security variables. Unfortunately, these frameworks cannot be applied to verify JavaScript programs because they highly rely on underlying type structure and JavaScript is a weak type language.

Although ECMAScript (Ecm09) provides a 200 page standard for explaining the syntax and semantics of JavaScript in the form of prose and pseudocode, this standard is too informal to be a foundation of formal verification.

Maffeis et al. (MMT08) presented a 30 pages standard that conforms to the ECMAScript standard with an abstract syntax of core JavaScript and a set of operation semantics rules. Guha et al. (GSS10) introduced a drastically different way to focus on the core of JavaScript and desugar the syntax of the language into a λ_{JS} semantics. Their semantics are more conventional and simpler than the semantics presented by

Maffeis et al, for example they use substitution instead of scope objects.

Yu et al. (YCIS07) proposed a framework to implement verification based on another subset of JavaScript, *CoreScript*. *CoreScript* is an imperative subset of JavaScript without the statements on function operations. However, the operational semantics they provided are inappropriate for safety proofs.

In fact, there are various subset languages proposed as a foundation of JavaScript language verification. Anderson et al. (AGD05) developed a substantial subset JS_0 for building up a type inference system. But their subset does not contain *prototype* and *function* features. Heidegger and Thiemmna (HT09; JMT09) included these two essential features, but omit assignment statement into their new type system. For constructing a subset language in a formal framework, there is a notable tradeoff between the ability to model all behaviours of a program and the need for rigour.

Since separation logic (LHQ08; CDNQ08; CDOY09; CDNQ10; DOY06; NDQC07) has proven an effective formalism for the analysis of memory-manipulating programs, Middelkoop et al. (MHK04) proposed a separation logic based framework for class-based languages, such as Java . In 2012, Gardner et al. (GMS12) produced a program logic for reasoning about a broad subset of JavaScript using separation logic. Their big-step operational semantics follows the work from Maffeis et al. (MMT08) in 2008.

2.4 Summary

This chapter first surveys the features of JavaScript. These features show the flexibility of the language in terms of dynamic features, essentially including implicit type conversion, object field modification on the fly, function object, prototype inheritance, and scope chain. Then it reviews the client-side applications of JavaScript, particularly in the aspect of third party applications. Then, safety issue and possible solutions are discussed. However, there are several problems with the existing solutions. One of the

common problems is that these solutions are only able to solve one particular issue. A decent solution needs to ensure the "quality" of a JavaScript application. Therefore, a formal verification framework on JavaScript is in need. A discussion on a number of existing frameworks in Table 2.3 shows that it is still an open research problem on formally modelling the semantics of JavaScript, reasoning about behaviours of the programs, and providing a formal framework to verify the functional correctness and safety of JavaScript programs. More detailed discussions are shown below:

- Since the safety issue is caused by maliciously manipulating *this* variable, most solutions focus on eliminating the variables that are either unsafe or grant uncontrolled access to a host page or that contribute to poor code quality. However, these solutions have proposed in the cost of sacrificing certain expressiveness of the language.
- The existing JavaScript formal frameworks studies merely concentrate on modelling a few features of JavaScript. Therefore, a comprehensive framework which is not only able to model the essence of the language but also the ability of verifying functional correctness of the programs is required.
- The research in separation logic shows the ability as a foundation of a formal framework to model the essential and flexible features of JavaScript. Most studies in separation logic based JavaScript formal framework, are focusing on verifying functional correctness of JavaScript programs. Middelkoop et al. (MHK04) stated that the main challenge lies in constructing operational and axiomatic semantics rules for the function invocation feature. Therefore, a elegant framework for verifying both functional correctness and safety of the programs is in needed.
- Another problem of many frameworks is that they ignored soundness proof. Huth and Tryan (HR04) stated that a formal framework must prove soundness property. Therefore, soundness proof of a proposed framework is in need.

Accordingly, this thesis will propose a comprehensive and elegant formal framework, which focuses on verifying functional correctness and safety properties of JavaScript programs. The framework is built on the underlying semantics of separation logic.

		Frameworks and Solutions												
		ADSafe	And05	Yu07	Jen09	Chu09	Sax10	Guh10	Dar12	Maf09	Rei07	Gua09	Dew10	Gar12 JS^{t}_{sl}
Features														
Obj Creation	Obj Literal	✔	✘	✘	✔	✔	✘	✔	✘	✔	✘	✘	?	✔
	Obj ”new” Crt.	✔	✘	✘	✔	✔	✘	✔	✘	✔	✘	✘	?	✔
Function	Function Declar.	✘	✔	✘	✔	✔	✔	✔	✘	✔	✔	✔	?	✔
	Function Expre.	✘	✘	✔	✘	✘	✘	✘	✘	✘	✘	✘	?	✔
	Method Call	✔	✔	✘	✘	✔	✘	✔	✔	✔	✔	✘	?	✔
	Global Call	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	?	✔
	Nested Func.	✔	✘	✘	✘	✘	✘	✔	✔	✔	✔	✔	?	✔
Field	Field Crt.	✔	✔	✘	✔	✔	✔	✔	✔	✔	✔	✔	?	✔
	Field Lookup	✔	✔	✘	✔	✔	✔	✔	✔	✔	✔	✔	?	✔
	Field Mutation	✔	✔	✘	✔	✔	✘	✔	✔	✔	✔	✔	?	✔
Variable	Global Assign.	✘	✔	✔	✘	✘	✔	✔	✔	✔	✘	✘	?	✔
	Local Assign.	✘	✔	✘	✘	✘	✘	✔	✔	✔	✘	✘	?	✔
	Expre. Return	✔	✘	✘	✔	✔	✔	✔	✔	✔	✔	✔	?	✔
With		✘	✘	✘	✘	✘	✘	✔	✔	✘	✔	✘	?	✔
Eval		✘	✘	✘	✘	✔	✘	✘	✘	✘	✔	✘	?	✔
”this” Keyword		✘	✔	✘	✘	✔	✘	✔	✔	✔	✔	✔	?	✔
Array		✔	✘	✘	✘	✘	✘	✘	✘	✔	✘	✘	?	✘
Iteration		✔	✘	✔	✔	✔	✔	✔	✔	✔	✔	✔	?	✔
Conditional		✔	✘	✔	✔	✔	✔	✔	✔	✔	✔	✔	?	✔
Prototype Inherit.		✔	✘	✘	✔	✔	✔	✔	✔	✔	✔	✔	?	✔
Scope Chain		✘	✘	✘	✘	✘	✔	✔	✔	✔	✘	✘	?	✔
Alias		✘	✔	✘	✘	✔	✘	✘	✔	✘	✔	✔	?	✘
Total No. of Features		13	10	5	11	15	10	18	16	18	15	12	?	20
(Total No. of Features/22) %		59%	45%	23%	50%	68%	45%	82%	73%	82%	68%	56%	?	90%
Problem														
Functional Correctness		✘	✘	✔	✘	✘	✘	✘	✘	✘	✘	✘	✔	✔
Memory Leak		✘	✘	✘	✘	✘	✘	✘	✔	✘	✘	✘	✘	✘
Capability Leak		✘	✘	✘	✘	✔	✘	✘	✔	✔	✔	✔	✘	✘
Safety Breach		✘	✘	✔	✘	✘	✘	✘	✔	✔	✔	✔	✘	✘
(Total No. of Problems/4) %		0%	0%	50%	0%	25%	0%	0%	75%	50%	50%	50%	25%	25%

Table 2.3: Comparisons of Frameworks for JavaScript

Chapter 3

JS_{sl} - A Subset of JavaScript

Inside every large program, there is a small program trying to get out. The job of formal methods is to elucidate the assumptions upon which formal correctness depends. – Tony Hoare (British computer scientist)

3.1 Introduction

Recent research shows that JavaScript can run improperly (Cro08; Crong; Fla11) in some circumstance. It indicates that the existing verification approaches (Flo67; Hoa69; Rey05) generally do not provide an elegant way to automatically verify JavaScript programs because of its dynamic features of subset of JavaScript (RLBV10). This chapter constructs a framework to model the dynamic features. Compared with the approaches discussed in Section 2.3.4, this framework defines an abstract core subset of JavaScript, JS_{sl} and proposes a sound separation logic based static axiomatic system to verify the functional correctness of JS_{sl} program. Firstly, the core language JS_{sl} (See Figure 3.1) is defined as the language used throughout the rest of this chapter. Secondly, a reliable formal operational semantics is presented with a stack and heap model. Thirdly, a set

of separation logic based axiomatic rules are constructed to support the functional correctness verification of JS_{sl} programs.

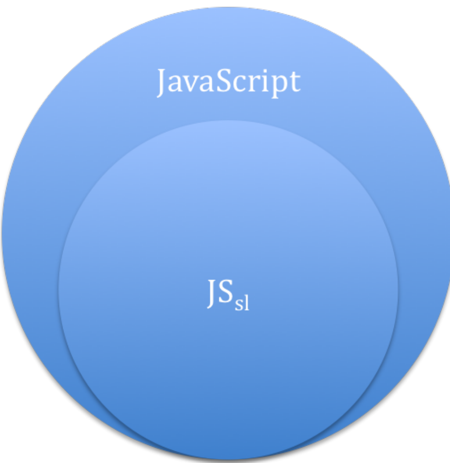

Figure 3.1: JavaScript v.s. JS_{sl}

3.2 The Language JS_{sl}

In this section, the target language JS_{sl} is defined. Being a prototype-based language, inheritance of JS_{sl} is performed via a process of cloning existing objects that serve as prototypes. Its syntax is formally defined in Figure 3.2.

A program in the JS_{sl} language consists of a sequence of *Statement*. Compared with the full version JavaScript language, this language has omitted some features, such as *array*, and *for loop*. The semantics of the language constructs follows the same conventions as in JavaScript, except for the global variable and local variable declaration. Other that that, the JS_{sl} behaves like the essences of a JavaScript program which is in Section 2.2.1, including the feature of prototype inheritance, function objects and object amplifying on the fly.

3.2.1 The Features and Conventions of JS_{sl}

The aim is to not only provide a realistic subset of JavaScript but also that it is manageable and feasible with respect to formalization and static verification. The main features of JS_{sl} are described as follows:

- Object fields[1] can be created dynamically on the fly.
- Support for prototype chain inheritance.
- Functions are treated as objects, a *Function Declaration* construct is not allowed in the language as they can be easily and often unintentionally turned into a *Function Expression*. Another reason is that it would cause significant and irreconcilable semantic difference (See Section 2.2.1.3).
- Support for parameter manipulation by value and reference. For the primitive data type parameters, the $copy{-}in$ algorithm is used to get a copy of the value inside the parameter and the update of the underlying parameter will not affect its original value. The reference type parameters perform according to the $copy{-}in{-}copy{-}out$ algorithm in which the update of the underlying variable also overwrite the original value. For example:

[1] In JavaScript, concept "property" is used instead of "field". We use *field* in conforming to the common terminology in OO area.

```
myNum = 10;
myObj = {name : "David", age : 12};
func changeVals(num, obj){
 var num = 0;
 var obj.name = "changed";
}
changeVals(myNum, myObj);
alert(myNum);                          // This returns integer 10
alert(myObj.name);                     // This returns changed
```

- Support distinction between global variables and local variables. A variable can be either declared explicitly or implicitly. The **var** keyword explicitly declares that the variable is a new identifier in the local scope that is the current function scope. Implicitly define a variable by simply referring to it without using the *var* keyword declares a global variable. Essentially, the explicit variable definition declares local variables, and the implicit definition declares global variables.

The other code conventions for JS_{sl} are based on JavaScript conventions. The object OProto is the global object *Prototype.Object* in JavaScript, it can be dereferenced to @proto that is an internal property in an execution context. FProto is the object for the root of @proto when the underlying object is a function type. @scope refers to a scope chain. The **this** variable is not supported as a statement in JS_{sl} but as an internal property of a function object whose value can be dereferenced to **this**. The statements *eval*, and *with* are not in the language JS_{sl}. In a literal object, more than one definition of the same data property is forbidden. Duplication of named parameters of a function is also not supported.

3.2.2 The Syntax of JS_{sl}

In Figure 3.2, there are expression Exp for variable, primitive type value, arithmetic or boolean operation, and function call. The $ExpFunc$ can be either an expression or a function expression. More details are given below:

- The arithmetic or boolean operation are $p(e_1, ..., e_n)$, for example:

  $add(x, y)$; // add variable x and y

  $times(x, y)$; // multiply variable x and y

  $ge(x, y)$; // compare variable x and y

- $F([x_1, ..., x_n])$ is defined as a returned function call in a function with optional parameters $x_1, ..., x_n$.

- **func** $[F]([x_1, ..., x_n])$ $\{c\}$ is a function expression abstraction with optional name F, optional parameters $x_1, ..., x_n$, and function body c.

- *skip* is for skip statement in the program execution process.

- x=ee is a global assignment that assigns ee to variable x. The variable x lives in the global scope.

- ***var*** $x = ee$ is a local variable assignment. All the local variables declared and assigned inside of a functions scope must be after the *var* keyword, and they are only visible in their defined function scopes. For example:

  $f =$ **func**$()\{$ var $x = 10;$ **return** $x;$ $\};$

  $alert(f())$; // This returns integer 10

  $alert(x)$; // This returns undefined

  The reason that the $alert(x)$ statement returns undefined is because x only lives in the scope of function f, and invisible from the global scope.

- $x = x'.f$ is a field lookup statement, the name of the field may either be declared initially or specified on the fly. For example:

  $obj = \{f_1 : 10, f_2 : 20\};$
  $obj.f_3 = \mathbf{func}()\{\mathbf{return}\ 30\};$ // This generates field f_3 *to obj on the fly*
  $x = obj.f_3();$ //This returns integer 30

  The field lookup follows the prototype chain which searches the current object for the field, if the field cannot be found in the scope of the current object, it searches along the prototype chain (See Section 2.2.1) until it reaches the global object *OProto*. In the case that the field cannot be retrieved from *OProto*, then an undefined value *undef* is returned.

- For the field mutation construct, $x.f{=}ee$, the value of field f is updated to ee when the field f is initially owned in the object record that is referred to by x or it is inherited from its prototype. In the case where the field f does not exist in the object record referred by x, a new value ee is added into the record dynamically. JS_{sl} supports arbitrary field modification of an object record on the fly.

- **return** e returns the value of the expression e.

- $x = x'.x_0([e_1, ..., e_n])$ and $x = x_0([e_1, ..., e_n])$ are two forms of function call that have a subtle difference: calling a function through an object and directly calling a function. Calling through an object $x = x'.x_0([e_1, .., e_n])$ implies that **this**[1] in the function refers to the receiver object x'; whereas when calling directly a function $x = x_0([e_1, .., e_n])$, **this** refers to the global object *OProto*. The internal variable

[1] The special variable **this** denotes the current object in scope. For example, when executing a function from a source object, **this** in the function refers to that enclosing object x', but when a function is executed independently, then variable **this** refers to the global object.

this is one of the execution context components, which is used to evaluate to the value of *ThisBinding* of the current execution context[1].

- The object literal statement $x = \{f_1{:}ee_1, .., f_n{:}ee_n\}$ creates a new object x that has a series of literal fields $f_1, .., f_n$ with a series of value $ee_1, .., ee_n$.

- The object creation statement $x = new\, x'()$ creates a new object x with its prototype field sets that are inherited from its prototype object x'. The newly created object x has an internal property @proto pointing to the OProto.

- The object creation function object $x = new\, x'([e_1, .., e_n])$ creates a new object x by invoking a function $x'([e_1, .., e_n])$. It creates a new object x through function object x' invocation. Note that the parameter list $e_1, ..., e_n$ is optional for the function. Compared with object creation statement, it has newly created object as a type of function with internal property @proto pointing to FProto instead of OProto.

- Statements sequence is *c ; c*. The program is a sequence of statements c separated by semicolons.

- Conditional statement is *if* (e) {c} *else* {c}. If e is true then execute statement before the *else*, otherwise execute the statement after the *else*.

- Iteration is *while* (e) {c}. While e is true then execute statement c.

3.3 Example

This section describes how a piece of JavaScript code transforms into JS_{sl} code using examples. Figure 3.3 is a piece of JavaScript program. It defines an object `obj` that has

[1] An execution context is purely a specification mechanism which is not necessary to correspond to any particular JavaScript interpreters.

$e \in Exp$	$::=$	x	Variable
	$\mid$	v	Primitive Value
	$\mid$	$p(e_1, ..., e_n)$	Arithmetic or Boolean
	$\mid$	$F([x_1, ..., x_n])$	Function
$ee \in ExpFunc$	$::=$	e	Exp
	$\mid$	$\textbf{func}\ [F]([x_1, ..., x_n])\ \{c\}$	[named] FuncExp
$c \in Statement$	$::=$	$\textbf{skip}$	Skip
	$\mid$	$x = ee$	GlobalAssignment
	$\mid$	$\textbf{var}\ x = ee$	LocalAssignment
	$\mid$	$x = x'.f$	FieldLookup
	$\mid$	$x.f = ee$	FieldMutation
	$\mid$	$\textbf{return}\, e$	Return
	$\mid$	$x = x'.x_0([e_1, ..., e_n])$	FuncCall
	$\mid$	$x = x_0([e_1, ..., e_n])$	FuncCall
	$\mid$	$x = \{f_1 : ee_1, ..., f_n : ee_n\}$	ObjLiteral
	$\mid$	$x = new\, x'()$	ObjCreation
	$\mid$	$x = new\, x'([e_1, .., e_n])$	ObjCrtFunc
	$\mid$	$c;\ c$	Sequencing
	$\mid$	$if\,(e)\ \{c\}\ else\ \{c\}$	Condtional
	$\mid$	$while\ (e)\ \{c\}$	Iteration
Identifiers			
$F \in FuncID$	$::=$	$F \mid F' \mid ...$	
$f \in FieldID$	$::=$	$f \mid f' \mid f_1 \mid x \mid ...$	
$b \in BooleanID$	$::=$	$True \mid False$	
$x \in VariableID$	$::=$	$x \mid x' \mid x_0 \mid ...$	
$v \in Variable$	$::=$	$int \mid str \mid null \mid undef$	
[...]	$::=$	$optional$	

Figure 3.2: Syntax of JS_{sl}

two fields, one of them (`f2`) is a function object. There is a conditional construct inside of function object `f3` that is nested in the function object `f2`. The JavaScript example of Figure 3.3 demonstrates the following features:

- Creation of object using object literal (line2 to 11),
- Creation of function object (line 5 to 8),
- Conditional Statement (line 6 and 7)
- Output of object (line 12, 13, 15 and 17),
- Field f_3 mutation on the fly (line 14),
- Object creation via function f_2.

Figure 3.4 presents the JS_{sl} program that is transformed from the JavaScript code in Figure 3.3 and conforms to the syntax of JS_{sl}. This transformation does not change the program semantics. Again, such a transformation aids to formalize and verify JavaScript in a realistic subset language. Note that the `alert` has no effect in the program, as it only produces an output.

3.4 Semantics for JS_{sl}

The structural operational semantics for JS_{sl} is a big-step semantics which shows transitions between machine configurations. Each machine configuration is a triple consisting of current program statement c, stack s and heap h. Before we approach to construct the operational semantics of JS_{sl}, we firstly define the semantic domains.

3.4.1 Semantic Domain

In the semantic domain, variables include the type of primitive variables and locations. A value is either a primitive value or a location. Primitive variables are type of

```
1  <script type="text/javascript">
2  var obj = {
3     f1: 1,
4     f2: function(n) {
5        var fn3 = function() {
6           if (n >= 10) { return 2;}
7           else { return 3;}
8        }
9        return fn3();
10    }
11 };
12 alert(obj.f2(11));        //2
13 alert(obj.f3);            //undefined
14 obj.f3 = 5;
15 alert(obj.f3);            //5
16 res = obj.f2(1);          //3
17 alert(res);
18 </script>
```

Figure 3.3: JavaScript Example

```
1  obj = {
2     f1: 1,
3     f2: func(n) {
4        var fn3 = func() {
5           if (ge(n,10)) { return 2 }
6           else { return 3}
7        }
8        var x = fn3();
9        return x
10    }
11 };
12 obj.f3 = 5;
13 res = obj.f2(1)
```

Figure 3.4: JS_{sl} Example

integer number, boolean value, string, and the set consisting of undefined and null. Locations correspond to object identifiers that can be partially viewed as memory addresses. Note that, a location ℓ can be a nullable memory address in some context. Objects are either a literally declared object which is a finite mapping from field identifiers to values or function expressions.

$$\mathbf{Value} = \mathbf{Prim} \cup \mathbf{Loc}$$

$$\mathbf{Prim} = \mathbf{Int} \uplus \mathbf{Bool} \uplus \mathbf{Str} \uplus \{\mathbf{undef}, \mathbf{null}\}$$

The operation semantics for the JS_{sl} language provides the meanings of the language program constructs that transforms from initial state to final state in a mathematically rigorous way. The program state δ is a pair consisting of stack s and heap h:

$$s, h \in \mathbf{State} = \mathbf{Stack} \times \mathbf{Heap}$$

- Stack s. Stacks are modelled as finite and stackable [1] mappings from variables to values. **Stack** denotes a partial function from variables (object identifications) to values or locations with a finite domain:

  $$s \in \mathbf{Stack} = \mathbf{Var} \rightarrow_{sfin} \mathbf{Value} \cup \mathbf{Loc}$$

  The semantics for the evaluation of an expression e in the stack is defined as *s(e)* where the expression e is associated with the current activated stack environment. It is defined as below:

  $$s(e) = \mathbf{Stack} \rightarrow \mathbf{Prim} \cup \mathbf{Loc} \cup \{\mathbf{error}\}$$

[1] Stackable mapping means a variable x might occur more than once, for example f: A $\rightarrow_{sfin}$ B denotes finite stackable mapping A to B, but it only updates the most recently included mapping of A by B.

Note that the evaluation of an expression in the stack may lead to a normal value (**Prim**) or a location (**Loc**) or an error (**error**). The **error** occurs while the statement or expression is being executed.

- Heap h. Heaps are modelled as finite partial mapping from locations to records. As shown below:

$$h \in \mathbf{Heap} = \mathbf{Loc} \rightharpoonup_{fin} \mathbf{Record}$$

Where a record is a reification [1] of an object in the heap. Records are the union of a finite mapping from fields to values and functions. Note that the functions in the records include parameters and function body. As shown below:

$$r \in \mathbf{Record} = (\mathbf{Var} \rightarrow_{fin} \mathbf{Value}) \cup \mathbf{Func}$$

Note that a direct written-form for records as $[n_1 : v_1, \ldots]$ is adopted, which describes that the record has the field n_1 with the value v_1, etc.

Furthermore, one of the implicit fields in a record for objects is @proto whose value is a reference to the record representing the prototype of this object. In other words, @proto could lead to the prototype of the underlying request object. In the prototype chain, the location of the root object OProto is defined as $\mathsf{loc_{op}}$.

In addition, "functions as objects" are adopted in JS_{sl} , a function **Func** can be stored in objects as a field. It contains three sub-fields: **body**, **params**, and @proto, as shown below:

$$\begin{aligned}\mathbf{Func} \quad = \quad & \{[\mathbf{body} : c, \mathbf{params} : (x_1, .., x_n), @\mathsf{proto} : \mathsf{loc_{op}}] \\ & \mid \ c \in \mathbf{ProcBody} \wedge n \in \mathbb{N}\}\end{aligned}$$

[1] The reification of an object refers to the explicit data model or other object created in the current object context. In my thesis, record reification involves finding a more concrete representation of the abstract object data type used in a formal specification, such as the primitive variables, functions and other objects.

where $(x_1, .., x_n)$ are parameters of the function, and c is the function body belonging to the set **ProcBody**. Here the value of @proto is taken as $\mathsf{loc_{op}}$ for every function object for brevity[1]. We use $\mathbb{N}$ to denote the set of natural numbers.

The semantics for the evaluation of an expression e in the heap is defined as $h(\ell)(e)$ where ℓ is a location that is in the range of the stack and e is associated expression in the heap. It is defined as below:

$$h(\ell)(e) = \begin{cases} \textbf{Heap} \rightarrow \textbf{Record} \\ \textbf{Heap} \rightarrow (\textbf{Prim} \cup \textbf{Loc}) \cup \{\textbf{error}\} \end{cases}$$

It is possible that there is more than one Stack and Heap memory cell in the Heap pool. Thus, an expression e in the heap pool can either be evaluated to be a record that is pointing to another heap cell or a union of a primitive value, location and error when the e is a local variable.

3.4.2 Operational Semantics

This section develops the operational semantics of the JS_{sl} language, which defines the behaviour of the program in terms of a set of transition relations and inference rules. The specification $\frac{A_1\ A_2\ ...\ A_n}{B}$ means that if the expression $A_1, A_2, ..., and\ A_n$ are true, then the program configuration B will update with the transitions. The configuration B is defined in the form of:

$$c, (s, h) \rightarrow (s', h') \quad \text{or}$$
$$c, (s, h) \rightarrow \bot$$

where c is the underlying program construct in the current context, (s, h) defines a program state δ, the transition starts from the initial state (s, h) and terminate at state (s', h') after finishing the execution of c. The $\bot$ represents abortion when the execution

[1] In JavaScript, the @proto field of every function objects refers to Function.Prototype. This simplification does not alter the semantics.

fails. The symbols δ and $\bot$ are used to represent a program state in the concrete semantic model.

The full set of transition rules for all program statements, including rules for variable assignment (see Figure 3.5), rules for operations on fields(see Figure 3.6), rules for function invocation (see Figure 3.7), rules for object creation (see Figure 3.8), and rules for control structures (see Figure 3.9).

In the case where the evaluation of a statement reaches **error**, the final program state would end at abortion ($\bot$). In the final state, any expression inside of the operation [ ] is an updated part to the initial state after an execution of a statement c. Furthermore, the operation $s[x \mapsto v]$ "pushes" the variable x onto s with the value v. The operation $s' = s \backslash \chi$ removes a set of variables χ from the domain of s. That is $\mathsf{dom}(s \backslash \chi) = \mathsf{dom}(s) \backslash \chi$ and $(s \backslash \chi)(x) = s'(x)$, for any $x \in \mathsf{dom}(s) \backslash \chi$. For example, $s' = s \backslash \{x_0\}$ removes the variable x_0 from the domain of s, and $(s \backslash x_0)(x) = s'(x)$ means that it evaluates the variable x from the domain of s', where $x \in \mathsf{dom}(s) \backslash x_0$. Similarly, the operation $h[\ell \mapsto r]$ extends the heap with a cell $\ell \mapsto r$. Note that, the expression $h[\ell \mapsto r[x \mapsto v]]$ evaluates the variable x in the record r in the heap h, and returns the value v.

In addition, there are various cases that may lead the execution of a statement to fail such as when the evaluation of an expression gives an error or the variable to be assigned to does not exist. The expression **or else** is used to combine several different failure cases. For example, A **or else** B defines that $A \vee (\neg A \wedge B)$.The **or else** expression is left associative:

$$A \;\; \textbf{or else} \;\; B \; \textbf{or else} \;\; C = (A \;\; \textbf{or else} \;\; B) \; \textbf{or else} \;\; C.$$

Figure 3.5 defines the inference rules for global variable assignment and local variable assignment. The full set of rules are described as follows:

- In [op-skip], both the stack and heap are unmodified.

$$\frac{}{\mathsf{skip}, (s,h) \rightarrow (s,h)} \quad \textbf{[op-skip]}$$

$$\frac{s(e) = v}{x = e, (s,h) \rightarrow (s[x \mapsto v], h)} \quad \textbf{[op-glob-assign1]}$$

$$\frac{s(e) = \mathbf{error}}{x = e, (s,h) \rightarrow \bot} \quad \textbf{[op-glob-assign1-abt]}$$

$$\frac{\ell \in dom(h) \quad h(\ell)(e) = v}{\mathbf{var}\ x = e, (s,h) \rightarrow (s, h[\ell \mapsto r[x \mapsto v]])} \quad \textbf{[op-local-assign1]}$$

$$\frac{\ell \notin dom(h) \ \ \textbf{or else} \ \ \ell = \mathbf{null} \ \ \textbf{or else} \ \ h(\ell)(e) = \mathbf{error}}{\mathbf{var}\ x = e, (s,h) \rightarrow \bot} \quad \textbf{[op-local-assign1-abt]}$$

$$\frac{s(x) = \ell \quad \ell \neq \mathbf{null} \quad h(\ell) = r \qquad r = [\mathbf{body} : c, \mathbf{params} : (x_1..x_n), @\mathsf{proto} : \mathsf{loc_{op}}]}{x = \mathbf{func}[F](x_1, ..., x_n)\{c\}, (s,h) \rightarrow (s[x \mapsto \ell], h[\ell \mapsto r])} \quad \textbf{[op-glob-assign2]}$$

$$\frac{s(x) = \mathbf{error} \ \ \textbf{or else} \ \ \ell = \mathbf{null}}{x = \mathbf{func}[F](x_1, ..., x_n)\{c\}, (s,h) \rightarrow \bot} \quad \textbf{[op-glob-assign2-abt]}$$

$$\frac{\ell_0 \in dom(h) \quad \ell \neq \mathbf{null} \quad h(\ell_0)(x) = \ell \qquad h(\ell) = [\mathbf{body} : c, \mathbf{params} : (x_1..x_n), @\mathsf{proto} : \mathsf{loc_{op}}]}{\mathbf{var}\ x = \mathbf{func}[F](x_1, ..., x_n)\{c\}, (s,h) \rightarrow (s, h[\ell_0 \mapsto r[x \mapsto \ell]])} \quad \textbf{[op-local-assign2]}$$

$$\frac{\ell \text{ is not in h} \ \ \textbf{or else} \ \ \ell = \mathbf{null} \ \ \textbf{or else} \ \ h(\ell)(x) = \mathbf{error}}{\mathbf{var}\ x = \mathbf{func}[F](x_1, ..., x_n)\{c\}, (s,h) \rightarrow \bot} \quad \textbf{[op-local-assign2-abt]}$$

Figure 3.5: Operational Semantics for Variable Assignments

$$\frac{x' \in \mathsf{dom}(s) \quad s(x') \in \mathsf{dom}(h) \quad f \in dom(h(s(x'))) \quad h(s(x'))(f) = v}{x = x'.f, (s, h) \rightarrow (s[x \mapsto v], h)} \quad \textbf{[op-lookup-field]}$$

$$\frac{\begin{array}{c} x' \in \mathsf{dom}(s) \quad s(x') \in \mathsf{dom}(h) \quad f \notin \mathsf{dom}(h(s(x'))) \\ h(s(x'))(\mathsf{@proto}) = s(x'') \quad x = x''.f, (s, h) \rightarrow (s', h') \end{array}}{x = x'.f, (s, h) \rightarrow (s', h')} \quad \textbf{[op-lookup-proto]}$$

$$\frac{\begin{array}{c} x' \in \mathsf{dom}(s) \quad s(x') \in \mathsf{dom}(h) \quad h(s(x')) = \mathsf{OProto} \\ f \notin \mathsf{dom}(\mathsf{OProto}) \end{array}}{x = x'.f, (s, h) \rightarrow (s[x \mapsto \mathsf{undef}], h)} \quad \textbf{[op-lookup-undef]}$$

$$\frac{x' \notin \mathsf{dom}(s) \ \textbf{ or else } \ s(x') \notin \mathsf{dom}(h)}{x = x'.f, (s, h) \rightarrow \bot} \quad \textbf{[op-lookup-field-abt]}$$

$$\frac{x \in \mathsf{dom}(s) \quad s(x) = \ell \quad \ell \in \mathsf{dom}(h) \quad h(\ell) = r \quad s(ee) = v}{x.f = ee, (s, h) \rightarrow (s, h[\ell \mapsto r[f \mapsto v]])} \quad \textbf{[op-mutate-field]}$$

$$\frac{\begin{array}{c} s(ee) = \textbf{error} \ \textbf{ or else } \ x \notin \mathsf{dom}(s) \\ \textbf{or else } \ s(x) \notin \mathsf{dom}(h) \ \textbf{ or else } \ h(s(x)) = \textbf{error} \end{array}}{x.f = ee, (s, h) \rightarrow \bot} \quad \textbf{[op-mutate-field-abt]}$$

$$\frac{\ell \in dom(h) \quad \ell \neq \textbf{null} \quad h(\ell)(e) = v}{\textbf{return } e, (s, h) \rightarrow (s, h[\ell \mapsto v])} \quad \textbf{[op-return]}$$

$$\frac{\ell \notin dom(h) \ \textbf{ or else } \ \ell = \textbf{null} \ \textbf{ or else } \ h(\ell)(e) = \textbf{error}}{\textbf{return } e, (s, h) \rightarrow \bot} \quad \textbf{[op-return-abt]}$$

Figure 3.6: Operational Semantics for Field Statements

- In [op-glob-assign1], v is the evaluation of expression e, the heap is unmodified.

- In [op-local-assign1], the variable x is declared in the local scope of a function body and has location ℓ in the heap. If ℓ is an existing nullable location, the local variable x is declared in ℓ with value v. The evaluation of the local assignment causes the stack to be unmodified and the heap is extended by $x \mapsto v$.

- In [op-glob-assign2], if ℓ is a nullable the location of variable x in the stack with a value of record r. A new record r is created in the heap. The heap cell stores the details of the function F. Thus, in the stack s, ℓ is bound to x. In the heap, r is allocated with a set of function body c, params $x_1, ..., x_n$ and @proto with its location $\mathsf{loc_{op}}$. Note that, $\mathsf{loc_{op}}$ is the location of OProto. This rule implies two cases: in the case of $l \in dom(h)$, it overwrites the location ℓ, whereas in the case of $\ell \notin \mathsf{dom}(h)$, it generates a new location ℓ.

- In [op-local-assign2], it is similar with the [op-glo-assign2] except that the statement is declared in an existing location ℓ_0 in the heap. If the function object x had location ℓ, thus the stack is unmodified, but the heap is extended by the cell of $x \mapsto \ell$.

In Figure 3.6, the rules for the object field operations are defined. The language JS_{sl} allows that all the program transitions on field operation are located in global scope rather than local scope except for the **return** statement. The full set of rules are described as follows:

- In [op-lookup-field], x' is in the domain of stack s and has value v in the heap. The value v of the request field f is bound to the variable x in the stack, the heap is unmodified.

- In [op-lookup-proto], in the case where the field f does not belong to the domain of the heap h, then it follows the prototype chain with @proto to retrieve its prototype

object x'' to evaluate f by [**op-lookup-field**] recursively.

- In [**op-lookup-undef**], the $undefined$ value is bound to x when the field f does not belong to the fields of the root object $OProto$ that can be followed by the prototype chain. The heap is unmodified.

- In [**op-mutate-field**], the existing field f of object x is updated, the stack is unmodified. In the heap, the record r that contains f mapping to the value of ee is bound to value of x. This rule can be applied to two cases. In the case where f does not exist in the object x, f is generated as a new field of x, and its value is set to be the evaluation of ee. In the case where the field f does exist in the object x, hence f will be automatically rewrote to have value v.

- In [**op-return**], v is the evaluation of expression e, the heap is unmodified.

In Figure 3.7, the rules for function expression and function invocation are defined. The language JS_{sl} allows these constructs to be only visible in global scope. In the following, the semantic rules for function expression and function invocation are specified.

- In [**op-fun-call-obj**], the object x has the requested function x_0. Invoking the function x_0 causes the execution of function body c that includes the evaluation of parameters $e_1, ..., e_n$, and the variable $this$ points to the object x'. After the execution of c, in the stack, s carries the original stack s and pushes the variable x to s with the evaluation of the variable **result** [1]. In the heap, h_1 is the heap that h' removes the variable $this$ and parameters. As the local function parameters and the variable $this$ should be not accessible from the outside of the function but also should be carried in the heap cell, therefore we have the heap h_2 to store the completed data after the invocation of function x_0.

[1]"result" is the function return expression. To evaluate such expression can solve the function return value.

$$\frac{\begin{array}{c} x' \in \mathsf{dom}(s) \quad s(x') \in \mathsf{dom}(h) \quad h(s(x')) = r \\ x_0 \in \mathsf{dom}(r) \quad r(x_0) = r' \quad r' \in \textbf{Func} \\ r'(\textbf{body}) = c \quad r'(\textbf{params}) = (x_1, .., x_n) \\ c, (s, h[\textbf{this} \mapsto h(s(x')), x_1 \mapsto h(e_1), .., x_n \mapsto h(e_n)]) \to (s', h') \\ s_1 = s[x \mapsto s'(\textbf{result})] \quad h_1 = h' \backslash \{\textbf{this}, x_1, .., x_n\} \quad h_2 = h \; h_1 \end{array}}{x = x'.x_0([e_1, ..., e_n]), (s, h) \to (s_1, h_2)} \quad \text{[op-fun-call-obj]}$$

$$\frac{\begin{array}{c} x' \notin \mathsf{dom}(s) \;\; \textbf{or else} \;\; s(x') \notin \mathsf{dom}(h) \;\; \textbf{or else} \\ h(s(x')) = r \quad x_0 \notin \mathsf{dom}(r) \;\; \textbf{or else} \;\; r(x_0) = r' \quad r' \notin \textbf{Func} \\ \textbf{or else} \;\; r'(\textbf{body}) = c \quad r'(\textbf{params}) = (x_1, .., x_n) \\ c, (s, h[\textbf{this} \mapsto h(s(x')), x_1 \mapsto h(e_1), .., x_n \mapsto h(e_n)]) \to \bot \end{array}}{x = x'.x_0([e_1, ..., e_n]), (s, h) \to \bot} \quad \text{[op-fun-call-obj-abt]}$$

$$\frac{\begin{array}{c} x' \in \mathsf{dom}(s) \quad s(x') \in \mathsf{dom}(h) \quad h(s(x')) = r \quad x_0 \notin \mathsf{dom}(r) \\ r(\mathsf{@proto}) = s(x'') \quad x = x''.x_0([e_1, ..., e_n]), (s, h) \to (s', h') \end{array}}{x = x'.x_0([e_1, ..., e_n]), (s, h) \to (s', h')} \quad \text{[op-fun-call-proto]}$$

$$\frac{\begin{array}{c} x' \in \mathsf{dom}(s) \quad s(x') \in \mathsf{dom}(h) \quad h(s(x')) = \mathsf{OProto} \\ x_0 \notin \mathsf{dom}(\mathsf{OProto}) \end{array}}{x = x'.x_0([e_1, ..., e_n]), (s, h) \to (s[x \mapsto \mathsf{undef}], h)} \quad \text{[op-fun-undef]}$$

$$\frac{\begin{array}{c} x_0 \in \mathsf{dom}(s) \quad s(x_0) \in \mathsf{dom}(h) \quad h(s(x_0)) = r \\ r \in \textbf{Func} \quad r(\textbf{body}) = c \quad r(\textbf{params}) = (x_1, .., x_n) \\ c, (s, h[\textbf{this} \mapsto loc_{go}, x_1 \mapsto h(e_1), .., x_n \mapsto h(e_n)]) \to (s', h') \\ s_1 = s'[x \mapsto s'(\textbf{result})] \quad h_1 = h' \backslash \{\textbf{this}, x_1, .., x_n\} \quad h_2 = h \; h_1 \end{array}}{x = x_0([e_1, ..., e_n]), (s, h) \to (s_1, h_2)} \quad \text{[op-fun-call-dir]}$$

$$\frac{\begin{array}{c} x_0 \notin \mathsf{dom}(s) \;\; \textbf{or else} \;\; s(x_0) \notin \mathsf{dom}(h) \\ \textbf{or else} \;\; h(s(x_0)) = r \quad r \notin \textbf{Func} \\ \textbf{or else} \;\; r(\textbf{body}) = c \quad r(\textbf{params}) = (x_1, .., x_n) \\ c, (s, h[\textbf{this} \mapsto loc_{go}, x_1 \mapsto h(e_1), .., x_n \mapsto h(e_n)]) \to \bot \end{array}}{x = x_0([e_1, ..., e_n]), (s, h) \to \bot} \quad \text{[op-fun-call-dir-abt]}$$

Figure 3.7: Operational Semantics for Function Invocation

- In [op-fun-call-proto], the object x' does not have such a function x_0. The prototype chain leads it to its prototype object x'' which actually does have x_0, then x_0 is evaluated by [op-fun-call-obj] recursively.

- In [op-fun-undef], all the objects (including OProto) in the prototype chain do not have function x_0 as their field. In the stack, s is directly extended to contain that x maps to $undef$. The heap is unmodified.

- In [op-fun-call-dir], it is similar with [op-fun-call-obj] but the variable $this$ is bound to global object Globj who has location loc_{go}, because the function invocation is occurred in the global scope.

The Figure 3.8 describes the varies of object creations semantic rules. Note that we exclude the scenario of creating object by function constructor, because the way of defining a function constructor is complied with function expression, and JS_{sl} supports function expression:

- In [op-obj-literal], we meet a statement which asks to create the object x by giving an object literal. In the stack, we extend it to contain the location ℓ of x. In the heap, we build a record which denotes initial values for each field, including the fields with corresponding values and @proto internal pointer points to OProto.

- In [op-obj-crt], the object x is generated by its prototype x' by keyword new. The location ℓ is not initially in the domain of the heap h but created on the fly. In the stack, we extend it to contain ℓ. In the heap, we extend it to include the variable @proto pointing to ℓ' that is the location of the variable x'.

- In [op-obj-crt-func], the object x can be created by using function $x'([e_1, ..., e_n])$ with keyword new. The location of the resulting object x is ℓ which is initially not part of the domain of heap h. The heap cell s_p contains the mapping from the variable $this$ to ℓ and the mapping from formal parameters to actual parameters. After

$$\frac{\begin{array}{c} s(ee_1) = v_1, ..., s(ee_n) = v_n \quad \ell \neq \mathbf{null} \\ r = [f_1 : v_1, ..., f_n : v_n, @\mathsf{proto} : \mathsf{loc_{op}}] \end{array}}{x = \{f_1 : ee_1, ..., f_n : ee_n\}, (s, h) \rightarrow (s[x \mapsto \ell], h[\ell \mapsto r])} \quad \textbf{[op-obj-literal]}$$

$$\frac{s(ee_i) = \mathbf{error} \text{ for some } i}{x = \{f_1 : ee_1, ..., f_n : ee_n\}, (s, h) \rightarrow \bot} \quad \textbf{[op-obj-literal-abt]}$$

$$\frac{x' \in \mathsf{dom}(s) \quad s(x') \in \mathsf{dom}(h) \quad s(x') = \ell' \quad \ell \neq \mathbf{null} \quad \ell' \neq \mathbf{null}}{x = new\, x'(), (s, h) \rightarrow (s[x \mapsto \ell], h[\ell \mapsto [@\mathsf{proto} : \ell']])} \quad \textbf{[op-obj-crt]}$$

$$\frac{x' \notin \mathsf{dom}(s) \ \textbf{ or else } \ s(x') \notin \mathsf{dom}(h) \ \textbf{ or else } \ \ell = \mathbf{null} \ \textbf{ or else } \ \ell' = \mathbf{null}}{x = new\, x'(), (s, h) \rightarrow \bot} \quad \textbf{[op-obj-crt-abt]}$$

$$\frac{\begin{array}{c} x' {\in} \mathsf{dom}(s) \quad s(x') {\in} \mathsf{dom}(h) \quad \ell \neq \mathbf{null} \quad \ell \text{ is new in h} \\ h(s(x')) \in \mathbf{Func} \quad h(s(x')) = r' \\ r' = [\mathbf{body} : c, \mathbf{params} : (x_1, ..., x_n), @\mathsf{proto} : \mathsf{loc_{op}}] \\ h_p = [this {\mapsto} \ell, x_1 {\mapsto} h(e_1), .., x_n {\mapsto} h(e_n)] \\ c, (s, h_p \ h[s(x) {\mapsto} [@\mathsf{proto} : s(x')]) \rightarrow (s', h') \\ s_1 = s'[x \mapsto \ell] \quad h_2 = h' \backslash \{this, x_1, .., x_n\} \quad h_2 = h \ h_1 \end{array}}{x = new\, x'([e_1, .., e_n]), (s, h) \rightarrow (s_1, h_2)} \quad \textbf{[op-obj-crt-func]}$$

$$\frac{x' \notin \mathsf{dom}(s) \ \textbf{ or else } \ s(x') \notin \mathsf{dom}(h) \ \textbf{ or else } \ \ell = \mathbf{null}}{x = new\, x'([e_1, .., e_n]), (s, h) \rightarrow \bot} \quad \textbf{[op-obj-crt-func-abt]}$$

Figure 3.8: Operational Semantics for Object Creation

the execution of the function body c, the resulting stack is s' and the resulting heap is h'. Similarly with [**op-fun-call-dir**], in the final state, the stack pushes the variable x to s' with the value of evaluation of the variable **result**. The heap h_1 removes the variable *this* and the parameters from h' to ensure that they are not accessible from the outside of function after the function returns. To reveal the data completion, the final heap h_2 is extended to include both h and h_1.

Due to the operational semantics of sequential, conditional, and iteration structures are standard and straightforward, as given in Figure 3.9, we omit the description for brevity.

3.5 An Axiomatic Framework for JS_{sl}

In this section, the *specification* language (also called *assertion* language) $Spec_{sl}$ is defined together with a set of symbolic execution rules. The specification language is developed based on separation logic, where the abstract program state is annotated by predicates, each of which describes either heap-insensitive statements or heap-sensitive statements. It is designed for reasoning about the programs that manipulate mutable data structures. The inference rules for JS_{sl} is also built up in the style of separation logic in terms of separation conjunction. The rationale for adopting separation logic is because it is designed for local reasoning about programs that manipulate heap-allocated data structures. The advantage of this includes that it can contribute to modelling of JS_{sl} programs that use the heap from an abstract point of views. The specification language explicitly captures the ownership of objects, which provides sufficient information to perform functional correctness property verification. In addition, it also manipulates the field modification on the fly which is another core feature of JS_{sl}.

$$\frac{c_1, (s, h) \to (s', h') \quad c_2, (s', h') \to (s'', h'')}{c_1;\ c_2, (s, h) \to (s'', h'')} \quad \textbf{[op-sequential]}$$

$$\frac{c_1, (s, h) \to \bot}{c_1;\ c_2, (s, h) \to \bot} \quad \textbf{[op-sequential-abt]}$$

$$\frac{s(b) = \mathsf{true} \quad c_1, (s, h) \to (s', h')}{\mathsf{if}(b)\ \{c_1\}\ \mathsf{else}\ \{c_2\}, (s, h) \to (s', h')} \quad \textbf{[op-conditional-true]}$$

$$\frac{s(b) = \mathsf{false} \quad c_2, (s, h) \to (s', h')}{\mathsf{if}(b)\ \{c_1\}\ \mathsf{else}\ \{c_2\}, (s, h) \to (s', h')} \quad \textbf{[op-conditional-false]}$$

$$\frac{s(b) = \mathbf{error}}{\mathsf{if}(b)\ \{c_1\}\ \mathsf{else}\ \{c_2\}, (s, h) \to \bot} \quad \textbf{[op-conditional-abt]}$$

$$\frac{s(b) = \mathsf{false}}{\mathsf{while}\ (b)\ \{c\}, (s, h) \to (s, h)} \quad \textbf{[op-iteration-false]}$$

$$\frac{s(b) = \mathsf{true} \quad c;\ \mathsf{while}\ (b)\ \{c\}, (s, h) \to (s', h')}{\mathsf{while}\ (b)\ \{c\}, (s, h) \to (s', h')} \quad \textbf{[op-iteration-true]}$$

$$\frac{s(b) = \mathbf{error}}{\mathsf{while}\ (b)\ \{c\}, (s, h) \to \bot} \quad \textbf{[op-iteration-abt]}$$

Figure 3.9: Operational Semantics for Control Structures

3.5.1 Specification Language for JS_{sl}

To capture the desired level of program correctness for JS_{sl} programs, logical operations are taken from separation logic to specify heap-allocated objects that can be used. The specification language uses the pre-condition and post-condition that are illustrated by logic formulas to describe "states" consisting of a stack and a heap. The meaning of the pair of pre-condition and post-condition is that if the execution of the program statement in the context of a program state satisfies its pre-condition, and if it terminates, then it terminates in another state that satisfies the corresponding post-condition. Otherwise, if the program state does not satisfy the post-condition, the verification fails and an error is reported.

According to the concrete heap model (see Section 3.4.1), a fixed finite collection fields (variables), and a disjoint set of locations, variables of non-addressable values are defined as below:

$$\begin{aligned} \mathbf{Heap} &= \mathbf{Loc} \rightharpoonup_{fin} (\mathbf{Var} \rightarrow_{fin} \mathbf{Value} \cup \mathbf{Func}) \\ \mathbf{Stack} &= \mathbf{Var} \rightarrow_{sfin} \mathbf{Value} \cup \mathbf{Loc} \end{aligned}$$

The framework adopts the partial correctness semantics of Hoare logic with tight interpretation. Tight interpretation is another crucial aspect of separation logic, which defines that well-specified programs do not go wrong. According to such interpretation, our language for reasoning about the heap model applies a certain pure (heap insensitive) and spatial (heap sensitive) assertions to describe the symbolic heap (abstract state). A symbolic heap Δ is a pair $\Pi \parallel \Sigma$ where Π is essentially a conjunction separated sequence of pure formula, and Σ is a $*$ separated sequence of partial formula. Therefore, we have a valid specification $\{\ \Delta_1\ \}\ \mathbf{c}\ \{\ \Delta_2\ \}$ ensure that command **c** should not encounter any memory faults when it starts in a program state satisfying Δ_1. The tight interpretation also requires the pre-condition Δ_1 of a statement to guarantee that all memory locations can be accessed by the execution of the statement, except for the freshly allocated ones

that are allocated in advanced. A memory location x is allocated with points-to relation, such as $x \mapsto _$ represents location x points-to somewhere in a heap cell.

In Figure 3.10, we show that a specification is a triple who is composed of an abstract pre-condition assertion Δ_1, command **c** and an abstract post-condition assertion Δ_2. For each symbolic heap Δ, the pure formulas include boolean values, a sequence of $\wedge$ and $\vee$ separated formulas, existential quantification variable, and expression assignment. The symbol "$\otimes$" represents the relationship of $\{=, >, <, \leq, \geq\}$. In the heap sensitive formulas, it essentially contains **emp** that denotes an empty heap, and the formula $x{\mapsto}[f_1 : e_1, .., f_n : e_n]$ denotes a heap-allocated object referred to by reference x, which contains fields $f_1, .., f_n$ whose values are $e_1, .., e_n$ where e and e_i $(i \in \mathbb{Z})$ represent expressions, and f_i $(i \in \mathbb{Z})$ denotes field names. The formula $\Sigma_1 * \Sigma_2$ (resp. $\Sigma_1 {-\!\!*} \Sigma_2$) denotes the separation conjunction (resp. separation implication) of two heap formula Σ_1 and Σ_2 that are two disjoined heap cells. For separation implication, Σ_2 is the updated heap cell with respect to Σ_1.

The semantics is given by a forcing relation $s, h \models M$ where s $\in$ **Stack**, h $\in$ **Heap**, and M can be a symbolic heap, a pure assertion, a heap assertion. This satisfiability is used to prove the soundness of a axiomatic framework in Section 3.5.3.

Furthermore, the full semantics is shown in Figure 3.11. For the pure Π, as noted in the last line of the figure, their semantics are defined with a specific notation A, which is preserved by the entailment prover that we use for soundness proving. Its definition is given in Figure 3.12

In our semantics, we allow a singleton heap formula $x{\mapsto}[f_1 : e_1, .., f_n : e_n]$ to specify partial fields of object x. The definition of heap disjointness is to allow partial objects being specified separately, we relax the usual disjointness definition as follows: $h_1 \# h_2$ iff $\mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset$ *or* $\forall\, \ell \in \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) \cdot \mathsf{dom}(h_1(\ell)) \cap \mathsf{dom}(h_2(\ell)) = \emptyset$. This allows us to represent a partial view of a heap-allocated object in our specifications. This flexibility does not cause any practical problems as our framework always main-

Specification	*Specsl*	$::= \{\ \Delta_1\ \}\ \mathbf{c}\ \{\ \Delta_2\ \}$
Abstract state	Δ	$::= \Pi\ \parallel\ \Sigma$
Pure formula	Π	$::= b \mid \Pi_1 \wedge \Pi_2 \mid \Pi_1 \vee \Pi_2 \mid \exists x.\Pi \mid x \circledcirc e$
Heap formula	Σ	$::= \mathbf{emp} \mid x \mapsto r \mid \Sigma_1 * \Sigma_2 \mid \Sigma_1 -\!\!* \Sigma_2 \mid \exists x.\Sigma$
Boolean	b	$::= \mathbf{true} \mid \mathbf{false} \mid x \mid b_1 = b_2$
Variable	x	$::= f \mid x' \mid x_i (i \in Z)$
Fields	f	$::= f' \mid f_i (i \in Z)$
Expression	e	$::= F \mid v \mid x \mid const \mid e_i (i \in Z)$
Record	r	$::= [f_1 : e_1, .., f_n : e_n] \mid e$

Figure 3.10: The Specification Language $Spec_{sl}$

tains a more complete view of an object via normalisation: $x \mapsto [f_1{:}e_1..f_n{:}e_n] * x \mapsto [f_{n+1}{:}e_{n+1}..f_{n+m}{:}e_{n+m}] \rightsquigarrow x \mapsto [f_1{:}e_1..f_{n+m}{:}e_{n+m}]$.

3.5.2 Inference Rules

This section defines inference rules for reasoning about JS_{sl} statements. Our inference rules abstractly capture the symbolic execution of these statements. The full list of rules can be found in Figure 3.13, Figure 3.14, Figure 3.15, and Figure 3.16.

The frame rule is derived from separation logic, which is shown as follows:

$$\frac{\{P\}\ c\ \{Q\} \quad modified(c) \cap vars(R) = \emptyset}{\{P * R\}\ c\ \{Q * R\}}\ [\underline{\textbf{sl-frame}}]$$

where $modified(c)$ denotes the program variables modified by c, and $vars(R)$ represents the set of free variables in the assertion R. Essentially, this [**sl-frame**] rule allows us to focus only in the parts of the heap that are actually manipulated by the execution of the command when we prove a specification (so called the memory footprint).

$$
\begin{array}{lll}
s,h \models \Pi \parallel \Sigma & \text{iff} & s \models \Pi \text{ and } (s,h) \models \Sigma \\
s,h \models \mathbf{emp} & \text{iff} & \mathsf{dom}(h) = \emptyset \\
s,h \models x \mapsto [f_1{:}e_1..f_n{:}e_n] & \text{iff} & \mathsf{dom}(h) = \{s(x)\} \text{ and} \\
 & & h(s(x)) = [f_1 : s(e_1), .., f_n : s(e_n)] \\
s,h \models x \mapsto F & \text{iff} & \mathsf{dom}(h) = \{s(x)\} \text{ and} \\
 & & h(s(x)) = [\mathbf{body} : c, \mathbf{params} : (...), @\mathsf{proto} : \mathsf{loc}_{\mathsf{fp}}] \\
s,h \models x \mapsto const & \text{iff} & \mathsf{dom}(h) = \{s(x)\} \text{ and} \\
 & & h(s(x)) = const, \text{ where const is OProto} \\
s,h \models \Sigma_1 * \Sigma_2 & \text{iff} & \exists h_1, h_2 \cdot h_1 \# h_2 \text{ and} \\
 & & h = h_1 * h_2 \text{ and } s, h_1 \models \Sigma_1 \text{ and } s, h_2 \models \Sigma_2 \\
s,h \models \Sigma_1 -\!\!* \Sigma_2 & \text{iff} & \forall h_1 \cdot (\mathsf{dom}(h_1) \cap \mathsf{dom}(h) = \emptyset \text{ and } s, h_1 \models \Sigma_1) \\
 & & \text{implies } s, h * h_1 \models \Sigma_2 \\
s,h \models \exists x \cdot \Sigma & \text{iff} & \exists r, h_1, h_2 \cdot h_1 \# h_2 \text{ and } h = h_1 * h_2, \\
 & & h_1 = [x \mapsto r] \text{ and } s, h \models \Sigma \\
s,h \models \Sigma & \text{iff} & h \models \Sigma \\
s \models \Pi & \text{iff} & s \models_A \Pi
\end{array}
$$

Figure 3.11: The Semantic Model for $Spec_{sl}$

$$
\begin{array}{lll}
s \models_A \textbf{true} & \text{iff} & \textbf{always} \\
s \models_A \textbf{false} & \text{iff} & \textbf{never} \\
s \models_A b_1 = b_2 & \text{iff} & s(b_1) = s(b_2) \\
s \models_A x & \text{iff} & s(x) = \textbf{true} \\
s \models_A x = e & \text{iff} & s(x) = s(e) \\
s \models_A x > e & \text{iff} & s(x) > s(e) \\
s \models_A x < e & \text{iff} & s(x) < s(e) \\
s \models_A x \leq e & \text{iff} & s(x) \leq s(e) \\
s \models_A x \geq e & \text{iff} & s(x) \geq s(e) \\
s \models_A \Pi_1 \wedge \Pi_2 & \text{iff} & s \models_A \Pi_1 \text{ and } s \models_A \Pi_2 \\
s \models_A \Pi_1 \vee \Pi_2 & \text{iff} & s \models_A \Pi_1 \text{ or } s \models_A \Pi_2 \\
s \models_A \exists x.\Pi & \text{iff} & s \models_A \Pi[v/x] \text{ for some } \mathbf{v}
\end{array}
$$

Figure 3.12: The Semantic Model for Pure Formula in $Spec_{sl}$

We defined a helper function $\mathsf{LV}(\Sigma)$ to describe the relationship between a variable and the heap formula:

$$\begin{array}{lcl} \mathsf{LV}(\mathbf{emp}) & ::= & \emptyset \\ \mathsf{LV}(x \mapsto r) & ::= & \{x\} \\ \mathsf{LV}(\Sigma_1 * \Sigma_2) & ::= & \mathsf{LV}(\Sigma_1) \cup \mathsf{LV}(\Sigma_2) \end{array}$$

Besides, the operation $(\exists x \cdot \Pi_1) \wedge \Pi_2$ states that there is a variable x in the pure formula Π_1. Similarly, the operation $(\exists x \cdot_1) * \Sigma_2$ states that there is a variable in the heap formula Σ_1. In our rules, the expression @proto is an internal property for each object that is pointing to the prototype object of the current one. It may return the location $\mathsf{loc_{fp}}$ (it has *const* value FProto) or $\mathsf{loc_{op}}$ (it has *const* value OProto). By following a prototype chain, OProto is the root object at the end of the chain. The value FProto is the root object at the end of chain when the object is a type of function.

The inference rules for the statement of variable assignment and field manipulation are given in Figure 3.13. The explanations for the full set rules are shown as follow:

- In [**sl-skip**], the pre-condition and post-condition are the same.
- In [**sl-glob-assign1**], every variable declaration and assignment in the global scope is located in the pure formula. Its occurrences in the pre-condition are replaced by expression e in the post-condition.
- In [**sl-local-assign1**], local variables are all declared in the local scope and located in the heap formula. Their occurrences in the pre-condition are replaced by the expression e in the post-condition.
- In [**sl-glob-assign2**], $x \mapsto r$ is a heap cell that is generated by creating function F expression.
- In [**sl-local-assign2**], the function expression x is declared inside of a existing function

$$\frac{}{\{\Pi \parallel \Sigma\}\mathbf{skip}\{\Pi \parallel \Sigma\}} \quad \text{[sl-skip]}$$

$$\frac{}{\{\Pi[e/x] \parallel \Sigma\}x = e\{\Pi \parallel \Sigma\}} \quad \text{[sl-glob-assign1]}$$

$$\frac{}{\{\Pi \parallel \Sigma[e/x]\}\mathbf{var}\ x = e\{\Pi \parallel \Sigma\}} \quad \text{[sl-local-assign1]}$$

$$\frac{r = [\mathbf{body} : c, \mathbf{params} : (x_1, ..., x_n), @\mathsf{proto} : \mathsf{loc_{op}}]}{\{\Pi \parallel \mathbf{emp}\}x = \mathbf{func}[F](x_1, ..., x_n)\{c\}\{\Pi \parallel \exists x \cdot x \mapsto r\}} \quad \text{[sl-glob-assign2]}$$

$$\frac{\Sigma \equiv \Sigma_0 * x_0 \mapsto r_0 \quad x_0 \in \mathbf{Func} \qquad r = [\mathbf{body} : c, \mathbf{params} : (x_1, ..., x_n), @\mathsf{proto} : \mathsf{loc_{op}}] \quad x \notin \mathsf{LV}(\Sigma)}{\{\Pi \parallel \Sigma\}\mathbf{var}\ x = \mathbf{func}[F](x_1, ..., x_n)\{c\}\{\Pi \parallel \Sigma * (\exists x \cdot x \mapsto r)\}} \quad \text{[sl-local-assign2]}$$

$$\frac{r = [..., f : v, ...]}{\{\Pi \parallel x' \mapsto r\}x = x'.f\{(\exists x \cdot \Pi) \wedge x = v \parallel x' \mapsto r\}} \quad \text{[sl-lookup-field]}$$

$$\frac{x' \notin \mathsf{LV}(\Sigma) \quad f \notin \mathsf{dom}(r) \qquad r(@\mathsf{proto}) = x'' \quad \{\Pi \parallel \Sigma\}x = x''.f\{\Pi' \parallel \Sigma'\}}{\{\Pi \parallel x' \mapsto r * \Sigma\}x = x'.f\{\Pi' \parallel x' \mapsto r * \Sigma'\}} \quad \text{[sl-lookup-proto]}$$

$$\frac{\Sigma \equiv x' \mapsto \mathsf{OProto} \qquad f \notin \mathsf{LV}(\Sigma)}{\{\Pi \parallel \Sigma\}x = x'.f\{\exists x \cdot \Pi \wedge x = \mathsf{undef} \parallel \Sigma\}} \quad \text{[sl-lookup-undef]}$$

$$\frac{r = [..., f : v, ...] \ \ \mathbf{or\ else} \ \ f \notin \mathsf{dom}(r)}{\{\Pi \parallel x \mapsto r\}x.f = ee\{\Pi \parallel x \mapsto r[f \mapsto ee]\}} \quad \text{[sl-mutate-field]}$$

Figure 3.13: Inference Rules for Variable Assignments and Field Statements

scope (the scope of x_0) located in the heap cell $x_0 \mapsto r_0$. a disjoint heap cell $x \mapsto r$ is generated by the creation of function expression x.

- In [sl-lookup-field], the pre-condition contains the existing object x' maps to record r, in the record r, we have the underlying object x with the corresponding value v. The pure formula is updated by $x = v$ in the post-condition.

- In [sl-lookup-proto], if the requested field f is not in the record of object x', but in another heap cell Σ, hence the @proto leads to its prototype object x'' and recursively applies [sl-lookup-field] rule. In the post-condition, the heap cell Σ is updated to Σ'.

- In [sl-lookup-undef], if the requested field f cannot be found in the record of object OProto, the post-condition only extends with $x = \mathsf{undef}$.

- In [sl-mutate-field], when the requested field f is in the record of object x or not in that record, under both circumstances the heap formula in the post-condition is updated by the heap cell of the variable $x \mapsto r$.

The inference rules refer to functions invocation are given in Figure 3.14:

- In [sl-fun-call-obj], when we call a function object x_0 from the object x', and x_0 can be fetched in the record of x', then the pure formula in the post-condition is updated by the variable x with value **res** that is the return value of calling the function x_0. Other certain updates in the heap formula from Σ to Σ_1.

- In [sl-fun-call-proto], if the requested function object x_0 is not reachable in the record of x', it follows @proto to its prototype object x'' and reach x_0 in somewhere by [sl-fun-call-obj]. The post-condition updates in the both pure formula and heap formula.

$$\frac{\begin{array}{c}\Sigma \equiv \Sigma_0 * x' {\mapsto} [x_0{:}x'', ..] * x'' {\mapsto} [\mathbf{body} : c, \mathbf{params} : (x_1..x_n), ...] \\ \Sigma_1 \equiv \Sigma * x_0 \mapsto [\mathbf{this} : x', x_1 : e_1, ..., x_n : e_n, ...] \\ \{\Pi \parallel \Sigma_1\} c \{\Pi_1 \parallel \Sigma_2\}\end{array}}{\{\Pi \parallel \Sigma\} x = x'.x_0([e_1, ..., e_n]) \{(\exists x \cdot \Pi_1) \wedge x = \mathbf{res} \parallel \Sigma_2\}} \quad [\text{sl-fun-call-obj}]$$

$$\frac{\begin{array}{c}x' \notin \mathsf{LV}(\Sigma) \quad x_0 \notin \mathsf{dom}(r) \quad r(\text{@}\mathsf{proto}) = x'' \\ \{\Pi \parallel \Sigma\} x = x''.x_0([e_1, ..., e_n]) \{\Pi' \parallel \Sigma'\}\end{array}}{\{\Pi \parallel x' \mapsto r * \Sigma\} x = x'.x_0([e_1, ..., e_n]) \{\Pi' \parallel \Sigma'\}} \quad [\text{sl-fun-call-proto}]$$

$$\frac{\begin{array}{c}\Sigma \equiv x' \mapsto \mathsf{OProto} \\ x_0 \notin \mathsf{LV}(\Sigma)\end{array}}{\{\Pi \parallel \Sigma\} x = x'.x_0([e_1, ..., e_n]) \{(\exists x \cdot \Pi) \wedge x {=} \mathsf{undef} \parallel \Sigma\}} \quad [\text{sl-fun-undef}]$$

$$\frac{\begin{array}{c}\Sigma \equiv (\Sigma_1 * x_0 \mapsto [\mathbf{body} : c, \mathbf{params} : (x_1...x_n), \text{@}\mathsf{proto}{:}\mathsf{loc_{fp}}, ...]) \\ \Sigma_1 \equiv (\Sigma * x' \mapsto [\mathbf{this} : \mathsf{loc_w}, x_1 : e_1, ..., x_n : e_n, ...]) \\ \{\Pi \parallel \Sigma_1\} c \{\Pi_1 \parallel \Sigma_2\}\end{array}}{\{\Pi \parallel \Sigma\} x = x_0([e_1, ..., e_n]) \{(\exists x \cdot \Pi_1) \wedge x = \mathbf{res} \parallel \Sigma_2\}} \quad [\text{sl-fun-call-dir}]$$

Figure 3.14: Inference Rules for Function Invocation

- In [sl-fun-undef], if the requested function object x_0 is not able to be fetched in record of OProto, thus in the post-condition, only the pure formula is extended by $x = \mathsf{undef}$.

- In [sl-fun-call-dir], in the case of directly invoking function x_0, the post-condition updates by function return value **res** in the pure formula and from Σ to Σ_2 in the heap formula.

A set of inference rules relevant to object creation are given in Figure 3.15, they cover the statements of new object creation via object literal and *new* keyword. Because *new* operation actually creates a prototype chain for resulting new object to inheritance fields, more explanations are shown below:

- In [sl-obj-crt-literal], the pre-condition starts from an empty heap part and allocates a new heap cell for the record of object x in the post-condition.

- In [sl-obj-crt-new], when the object x is created by x' via *new* operation, the post-condition extends its heap part by adding a new cell that specifies x' is the prototype of the object x.

- In [sl-obj-crt-fun], when the object x is created by function x' via *new* operation, the **this** internal property changed its value from the global variable *window* to the newly generated object x after the execution of the invoked function body c. Thus, the post-condition updates both pure formula and heap formula.

The inference rules for control structures, including sequential composition, conditional and while-loops, are standard as in Hoare logic, as shown in Figure 3.16, we omit their description for brevity:

$$\frac{r = [f_1 : e_1, ..., f_n : e_n]}{\{\Pi \parallel \mathbf{emp}\}\ x = \{f_1 : e_1, .., f_n : e_n\}\ \{\Pi \parallel \exists x \cdot x \mapsto r\}} \quad \textbf{[sl-obj-crt-literal]}$$

$$\frac{}{\{\Pi \parallel x' \mapsto r\}\ x = \mathit{new}\, x'()\ \{\Pi \parallel x' \mapsto r * (\exists x \cdot x \mapsto [\text{@proto} : x'])\}} \quad \textbf{[sl-obj-crt-new]}$$

$$\frac{\begin{array}{c}\Sigma \equiv (\Sigma_0 * x' \mapsto [\mathbf{body} : c, \mathbf{params} : (x_1..x_n), \text{@proto:loc}_{\text{op}}, \mathbf{this}\text{:loc}_{\text{w}}]) \\ \Sigma_1 \equiv \Sigma * (\exists x'' \cdot x'' \mapsto [\mathbf{this} : x, x_1 : e_1, ...x_n : e_n, ...]) \\ \{\Pi \parallel \Sigma\}c\{\Pi_1 \parallel \Sigma_2\}\end{array}}{\{\Pi \parallel \Sigma\}x = \mathit{new}\, x'([e_1, ..., e_n])\{\Pi_1 \parallel \Sigma_2\}} \quad \textbf{[sl-obj-crt-fun]}$$

Figure 3.15: Inference Rules for Object Creation

$$\frac{\{\Pi \parallel \Sigma\}c_1\{\Pi_1 \parallel \Sigma_1\} \quad \{\Pi_1 \parallel \Sigma_1\}c_2\{\Pi_2 \parallel \Sigma_2\}}{\{\Pi \parallel \Sigma\}c_1;\ c_2\{\Pi_2 \parallel \Sigma_2\}} \quad \textbf{[sl-sequential]}$$

$$\frac{\{\Pi \wedge b \parallel \Sigma\}c_1\{\Pi_2 \parallel \Sigma_2\} \quad \{\Pi \wedge \neg b \parallel \Sigma\}c_2\{\Pi_2 \parallel \Sigma_2\}}{\{\Pi \parallel \Sigma\}\mathbf{if}\ (b)\{c_1\}\mathbf{else}\ \{c_2\}\{\Pi_2 \parallel \Sigma_2\}} \quad \textbf{[sl-conditional]}$$

$$\frac{\{\Pi \wedge b \parallel \Sigma\}c\{\Pi \parallel \Sigma\}}{\{\Pi \parallel \Sigma\}\mathbf{while}\ (b)\{c\}\{\Pi \wedge \neg b \parallel \Sigma\}} \quad \textbf{[sl-iteration]}$$

Figure 3.16: Inference Rules for Control Structures

3.5.3 Soundness

We have defined the underlying operational semantics of our language JS_{sl} a set of inference rules for our axiomatic framework. We use those rules in conjunction to logically derive theorems. Our axiomatic framework is completely described with the soundness proof. In other words, the axiomatic framework is sound if each of its theorems is valid in every statements of the JS_{sl} language. Thus we have definitions for specification validity and soundness:

Definition 1 (Validity)**.** A specification $\{ \Delta_1 \}$ **c** $\{ \Delta_2 \}$ is *valid*, denoted $\models \{P\}c\{Q\}$, if and only if it is logically truth with respect to underlying operational semantics, denoted as, $\forall\ s, h.$ if $s, h \models \Delta_1$ and $c, (s, h) \rightarrow (s', h')$ for some s', h', then $s', h' \models \Delta_2$.

Definition 2 (Soundness)**.** Our verification framework for JS_{sl} is *sound* if all provable specifications under our axiomatic framework are indeed valid, denoted as, if $\vdash \{\Delta\}c\{\Delta_2\}$, then $\models \{\Delta_1\}c\{\Delta_2\}$.

Based on the definition above we have a theorem for our axiomatic framework:

Theorem 1. *Our axiomatic framework presented in this chapter is sound with respect to the underlying operational semantics.*

As is indicated by Definition 2 above, we need to show that, for any Δ_1, c, Δ_2, if $\vdash \{\Delta_1\}c\{\Delta_2\}$, then $\models \{\Delta_1\}c\{\Delta_2\}$. The proof can be accomplished by structural induction over c. For example, for the $lookup - field$ statement, we have following proof:

$$\frac{r = [..., f : v, ...]}{\{\Pi \parallel x' \mapsto r\} x = x'.f \{(\exists x \cdot \Pi) \wedge x{=}v \parallel x' \mapsto r\}} \quad \text{[sl-lookup-field]}$$

According to above definitions, the proof for rule [**sl-lookup-field**] can be written into:

$$\forall \text{ s,h. if } s,h \models \{(\exists x \cdot \Pi) \parallel x' {\mapsto} r\} \text{ and } (s,h) \to (s',h'), \text{ then } s',h' \models \{(\exists x \cdot \Pi) \wedge x{=}v \parallel x' {\mapsto} r\}$$

with respect to the following operational semantics:

$$\frac{x' \in \mathsf{dom}(s) \quad s(x') \in \mathsf{dom}(h) \quad f \in dom(h(s(x'))) \quad h(s(x'))(f) = v}{x = x'.f, (s,h) \to (s[x \mapsto v], h)} \quad \text{[op-lookup-field]}$$

In other words, the goal is to to prove the satisfiability of $s', h' \models \{(\exists x{\cdot}\Pi){\wedge}x{=}v \parallel x'{\mapsto}r\}$. The proof details is shown as follow:

Take any program state σ such that $(s,h) \models \{(\exists x{\cdot}\Pi) \parallel x'{\mapsto}r\}$. Under our operational semantics, we have $x = x'.f, (s,h) \to (s',h')$, where $(s',h') = (s[x \mapsto v], h)$. The goal is turned to prove the following satisfiability:

$$\text{s[x} \mapsto v], h \models \{(\exists x \cdot \Pi) \wedge x{=}v \parallel x'{\mapsto}r\}$$

The semantic domain in Figure 3.11 shows us that the goal can be turned to prove that:

$$\text{s[x} \mapsto v] \models (\exists x \cdot \Pi) \wedge x{=}v \text{ and } s[x \mapsto v], h \models x'{\mapsto}r$$

Due to $h = x' \mapsto r$, thus we will always have $h \models x'{\mapsto}r$. According to Figure 3.12, to prove that $s[x \mapsto v] \models (\exists x \cdot \Pi) \wedge x{=}v$, we only need to prove that:

$$\text{s[x} \mapsto v] \models \Pi \text{ and } s[x \mapsto v] \models (x = v)$$

Due to $s \models \Pi$, we will always have $s[x \mapsto v] \models \Pi$. As we know that $f \in \mathsf{dom}(h(s(x')))$ and $h(s(x'))(f) = v$, thus we will always have $s[x \mapsto v] \models (x = v)$. Note that the value v

could be s a primitive value or the reference if it is a reference type value (e.g. function type value). Therefore, we will always have $(s[x \mapsto v], h) \models \{(\exists x \cdot \Pi) \wedge x{=}v \parallel x'{\mapsto}r\}$. The soundness proof of the axiomatic framework JS_{sl} is done by structural induction over program statement and the details are in Appendix A on page 149.

3.6 Summary

We have described a new approach to verify the functional correctness of the JS_{sl} programs that have pointer-based data structures. We give the definitions of the target programming language, a substantial subset of JavaScript, JS_{sl}, which captures the core features and behaviours of JavaScript, such as prototype inheritance, function object, and automatic object amplifying on the fly. For the purpose to prove the soundness of our approach, we employ the operational semantics for our language and the semantic model for the specification language. Meanwhile, the specification language is introduced as a variant of separation logic and given a sound axiomatic framework. More detailed proof of the feasibility of our approach can be found from the experimental results in Chapter 5 and the appendix. This chapter is illustrated as the foundations to verify more properties of our language. In the next chapter, we turn to verify safety property in a more expressive subset ${JS^t}_{sl}$ program.

Chapter 4

JS^{t}_{sl} - A safe usage of this for JS_{sl}

Logical assertions can be used in formal verification in such a way that a rigorous standard is established for proofs about computer programs , including proof of correctness, safety. – R.W. Floyd (Computer scientist in Stanford University)

4.1 Introduction

One of the usages of JavaScript is constructing third party applications (*guest* codes) that embedded in a host page (*host* codes). The host page, a publisher, rents a portion of its web page to third party's network, such as publishing advertisement or applications. These third parties provide content that the browser displays on the user's screen. Malicious third parties could find a way that not only exploit the trust relationship between the publisher and users, but also inject malicious JavaScript code into a honest host page for attacking users. These attacks need to be prevented to improve the degree of safety for users.

The design decision was taken to focus on how to ensure that *guest* codes are safe

with respect to *host* page. Essentially, the unsafe interferences [1] between *guest* and *host* are caused by directly or indirectly manipulating the global object (the *window* [2] object). The way of accessing the *window* object is through the usage of **this** in a function execution context. For manipulating the use of **this**, JS_{sl} is extended to $JS^t{}_{sl}$ for rendering **this** expression. In $JS^t{}_{sl}$, a function treated as a field is called a method. This provides an interpretation for **this** variable as **this** can be bound to either the global object or a newly created object during the execution of the functions.

Consider the example JavaScript program in Figure 4.1. The code is used to show certain essential features of the language, including object structure, alias variable, function invocation. In Figure 4.1, an object *obj* is literally declared with two fields, x is a primitive type variable who has integer value 0, $setX$ is a function who is initiated as a method. In line 10, $window.x$ statement returns $undefined$ because the field x is not a global variable but a local variable for *obj*. The global object *window* can only reach the variables in the global scope, such as *obj* itself. However, it returns 10 in line 21 because the function is invoked and x has been implicitly attached as a global variable because of the execution of the function body statements. *obj.setX(10)* is applied in the line 12, **this** is bound to *obj* at this point. In the same figure, however, in the line 15, *f(90)* actually behaves that the execution makes **this** bound to the global object *window*, because the reference *obj.setX* is bound to f who is a global variable and the function invocation is under the global context. In fact, as mentioned in Section 2.2.1, **this** is an implicit variable to all JavaScript functions but its value behaves differently depending on the different type of function invocation with respect to **this**.

Since **this** could potentially point to the global object, the solution from many researches is to prohibit the use of **this** in *guest* code to restrict it to access the global object. Due to the script source codes can be arbitrary embedded within HTML file,

[1] An unsafe interference is about how the *guest* code achieves unauthorised accesses to the variables in the *host* page. For example, the *guest* code is able to manipulate the global object *window* of the *host* page.

[2] In the context of browser environment, the global object is *window* object.

```
1  <html>
2  ...  ...
3  <script type="text/javascript">
4  var obj = {
5     x : 0,
6     setX : function(n) { this.x = n; }
7  };
8
9  // window is the name of the global object in Web Browsers
10 window.x;                  //undefined
11
12 obj.setX(10);
13 obj.x;                     //10
14 f = obj.setX;
15 f(90);
16
17 //obj.x was not updated
18 obj.x;                     //10
19
20 //window.x was created
21 window.x;                  //90
22 </script>
23 ...  ...
24 </html>
```

Figure 4.1: Example of **this** variable manipulation in JavaScript

thus any guest codes can be mashed up with the host codes without causing semantic modification. It means that all the script codes can be semantically bound together under the $< script > ... < /script >$ tags. Therefore, a fragment of code may be directly or indirectly considered as malicious scripts as long as the value of **this** variable is innocently or maliciously altered to point to *window* object.

In this chapter we aim to provide a solution that allows the *guest* code to accommodate **this** variable. To analyse both innocent and malicious operations on **this** variable, we provide an elegant reachability graph analysis to show all the reachability relationships among the object, an axiomatic framework is constructed to verify **this** safe property. The reachability graph is another approach to verify programs in a visualised way. The reason why we adopted the reachability graph is that the Hoare Triple axiomatic system is a verification method in the mathematical and logical way, but the reachabil-

ity graph visualises the evolution of program states by steps, which can be assisting the readers to understand the meaning of the program status that is written by Hoare logic style specifications.

4.2 Example Analysis

To analyse the JavaScript program in Figure 4.1, firstly we transform it to a ${JS^t}_{sl}$ program in semantic-preserving way (see Figure 4.2). Secondly, the program is illustrated by our reachability graph analysis. Note that, further definition and explanation about reachability analysis are seen in Section 4.3. In this section, we present an overview of how the reachability graph analysis is able to extract the relationships among objects and discover the value modification of **this** explicitly or implicitly.

Furthermore, we view the program as a generator for data structures. And the data structures allocated in the heap are summarised by making a reachability graph. The relationships among the data structures are modelled as reachability between nodes, which one node corresponds to possible other nodes in the heap. The major issue is how to manipulate the heap cells to associate with which nodes, and with the growth of data structures how to manage the relationships for every heap cells. Our reachability graph contains nodes to represent program labels and edges to represent heap references. In Figure 4.3, we show the notations definition in our reachability graph. The circle node represents *global label node* (label node) that refers to a program label defined in the global context, the diamond node represents the primitive type value of variables that are located in the global scope only, the rectangle node represents *single object heap region node* who refers to a disjointed heap cell that might be referenced by an object. A heap region node consists of its reference address and a *reachability state* which indicates a set of objects that the given object can reach. The shaded summary heap region node represents the newest state [1] of the corresponding object node. The heap region node

[1]A newest state is a special heap region node for a function, which contains all the *reachability states*

```
1  obj = {
2     x : 0,
3     setX : func(n) { this.x = n }
4  };
5
6  n1= window.x;            //undefined
7
8  n2 = obj.setX(10);
9
10 n3 = obj.x;              //10
11
12 f = obj.setX;
13
14 n4= f(90);
15
16 n5 = obj.x;              //10
17
18 n6 = window.x            //90
```

Figure 4.2: Example of **this** variable manipulation in $JS^t{}_{sl}$

with M label on the top right represents the *multiple objects heap region node* that may be referenced by more than one object nodes. Note that we presume that the heap cell for a function invocation is automatically collected by Garbage Collector when the invocation is finished. And a new invocation operation located in the same cell where it might be collected previously. All the above nodes can be connected by $reference\ edge$ that represents the heap references. One object can reach another object if there exists an edge from the first object to the second object.

We analyse the example in Figure 4.2 by using reachability graph, the analysis begins from the line one that literately create object *obj* with field x and $setX$. Figure 4.4 presents the analysis results for it. At the beginning, *obj* is a label node that can be reached by *window* object at this point. The reference edge is established from the label node *obj* to the heap region node that is composed of the heap address *Addr_obj* and a *reachability state* set. The *reachability state* shows that label node *obj* can reach the heap region node *Addr_x* and *Addr_setX* respectively. The local field x is initialized by integer value 0 and another local field $setX$ is preprocessed as a method.

occurring in the original function

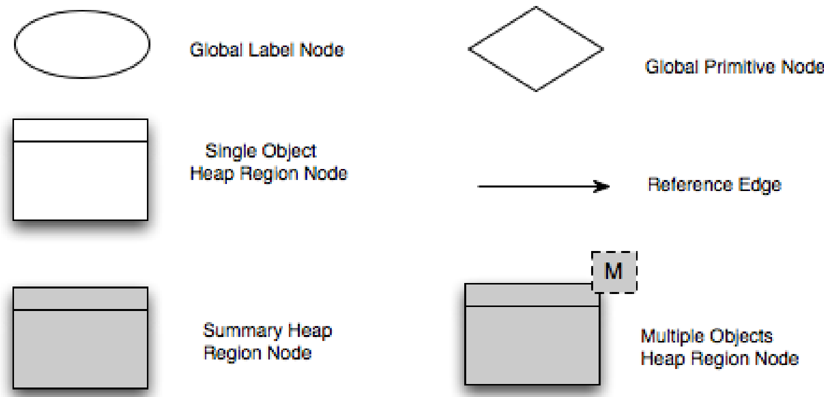

Figure 4.3: Reachability Graph Notations

Figure 4.5 presents the reachability graph immediately after line 6. A new label node has been created for n_1 who has *undefined* value. Besides, the heap region node *Addr_window* updates its *reachability state* with object n_1, and a new reference edge is generated from ⟨ *Addr_window*, n_1, *undefined* ⟩.

As in Figure 4.6, it represents the analysis result after line 8, the method *setX* is called by *obj* with certain argument, the new object n_2 is established with edge reference linked to *undefined* value. The function initialisation (function body and parameter) of *setX* is stored in a independent part of cell, it would be activated when it is invoked. A new shaped heap region node becomes visible in the graph since the execution of the function invocation statement *obj.setX*(10). Note that heap regions are shaded only when they are *summary node*. A new reference edge *setX* is generated and linked from the caller node *Addr_obj* to the callee summary node *Addr_setX*. The reference edge from *Addr_window* to *undefined* is updated to a set $\{n_1, n_2\}$. Meanwhile, the value of the node *Addr_x* is updated to 10 after we executed the function body of *setX*.

Figure 4.7 shows the reachability graph after the assignment of $n_3 = obj.x$. It creates reference edges ⟨ *Addr_window*, n_3, 10 ⟩. At this point, the *reachability state* for the summary node *Addr_setX* remains the latest value for primitive variable x who has

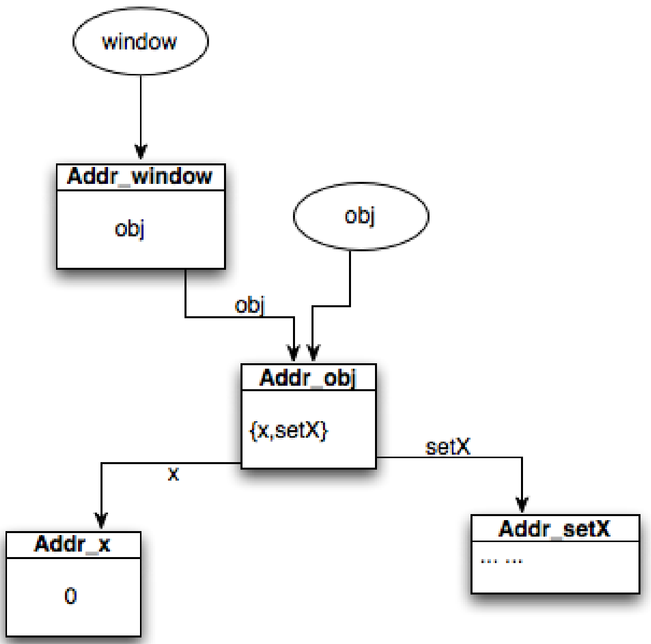

Figure 4.4: Reachability graph after line 1

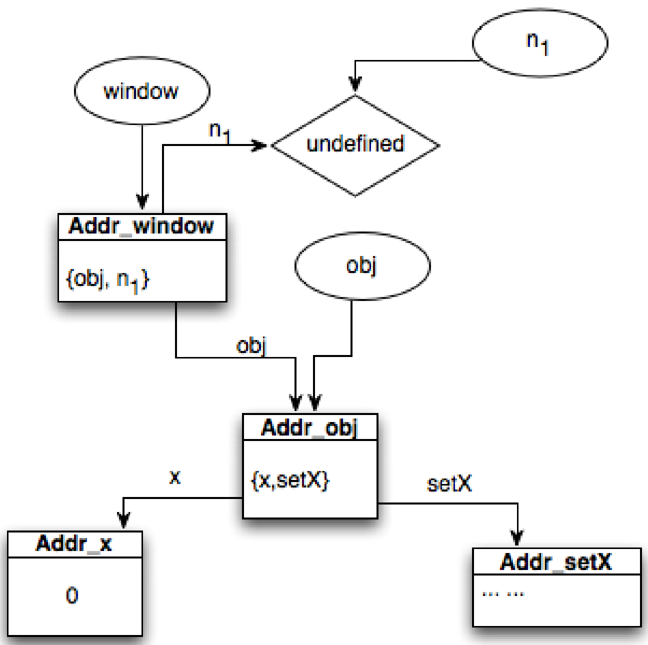

Figure 4.5: Reachability graph after line 6

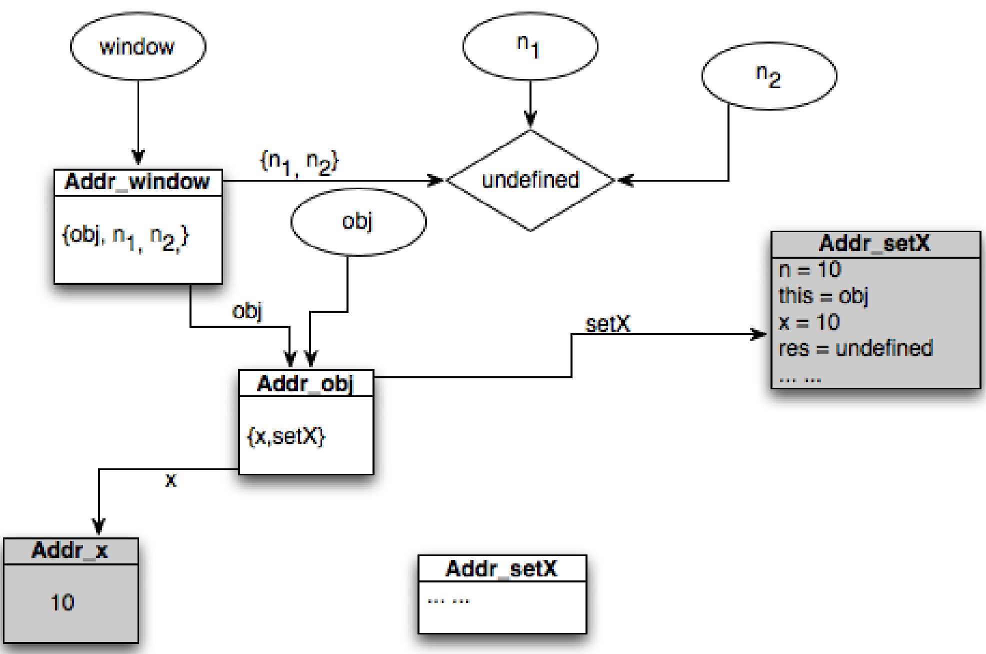

Figure 4.6: Reachability graph after line 8

integer value 10 which leads the return value for label n_3 to 10.

Figure 4.8 presents the reachability graph after line 12. A new label node is created for f and the *reachability state* for the node *Addr_window* is updated to include f. It also creates two reference edges $\langle$ *Addr_window*, f, *Addr_setX* $\rangle$ and $\langle$ f, *Addr_setX* $\rangle$ respectively.

In Figure 4.9, it presents the reachability graph result after that the line 14, method *setX* is invoked with parameter 90 in the context of global scope. Essentially, a new label node n_4 is created and the multiple object summary heap region node *Addr_setX*() is updated for altering the value of **this** to *window*. A few new reference edges are created, such as $\langle$ *Addr_window*, x, 90 $\rangle$, $\langle$ n_4, *undefined* $\rangle$,$\langle$ *Addr_window*, n_4, *undefined* $\rangle$.

In Figure 4.10, we have the reachability graph after line 16, it implies that the value of x that is reachable by *Addr_obj* remains 10, but the global variable x that is reachable by *Addr_window* has value 90. A label node is created for n_5 pointing to 10. Meanwhile,

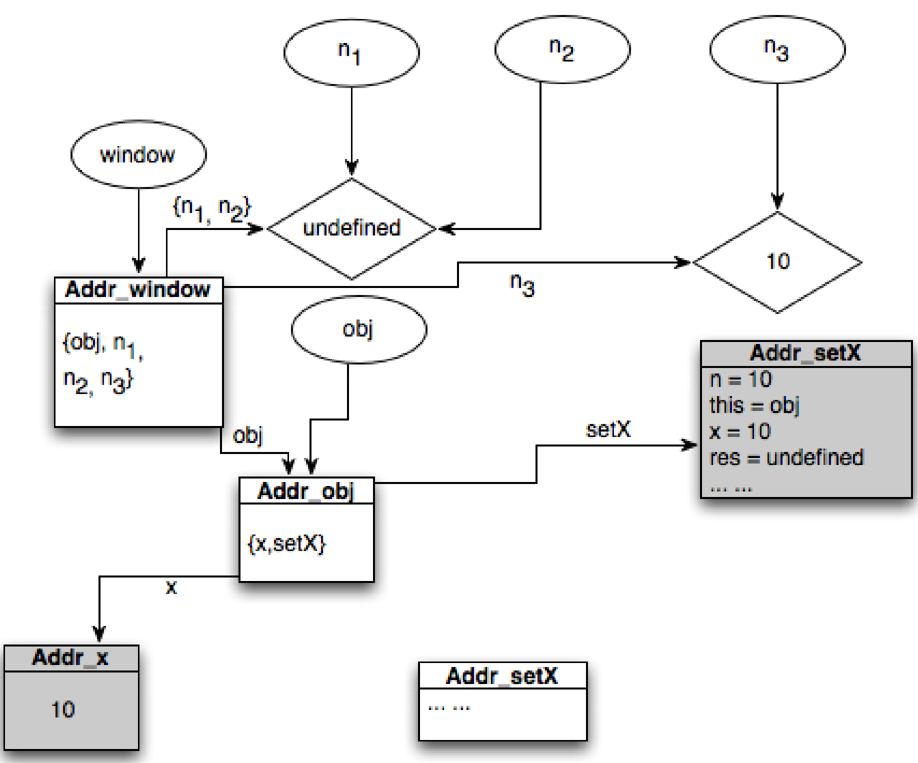

Figure 4.7: Reachability graph after line 10

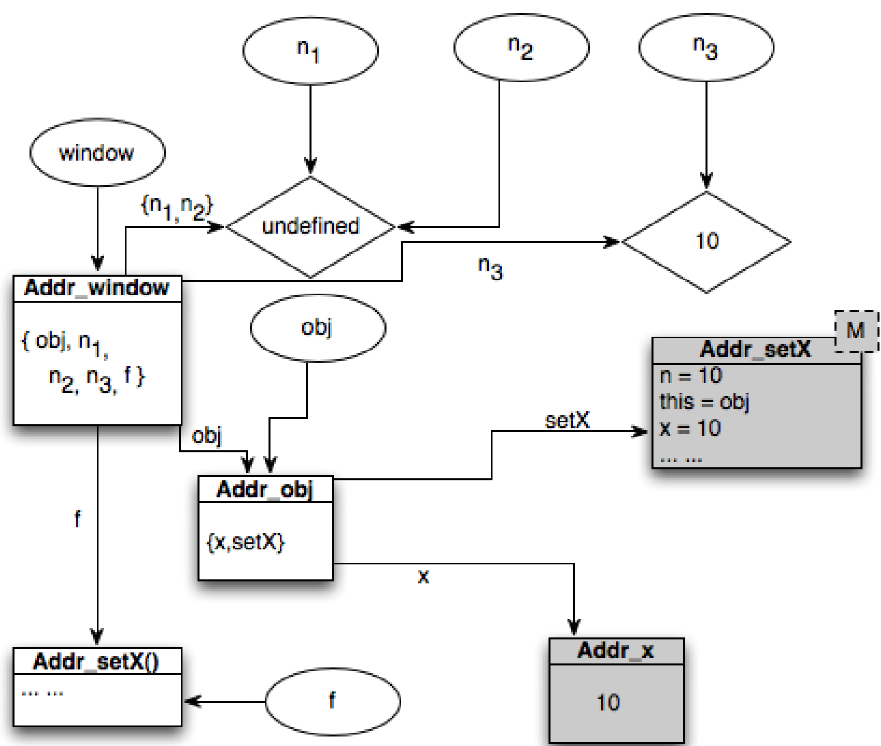

Figure 4.8: Reachability graph after line 12

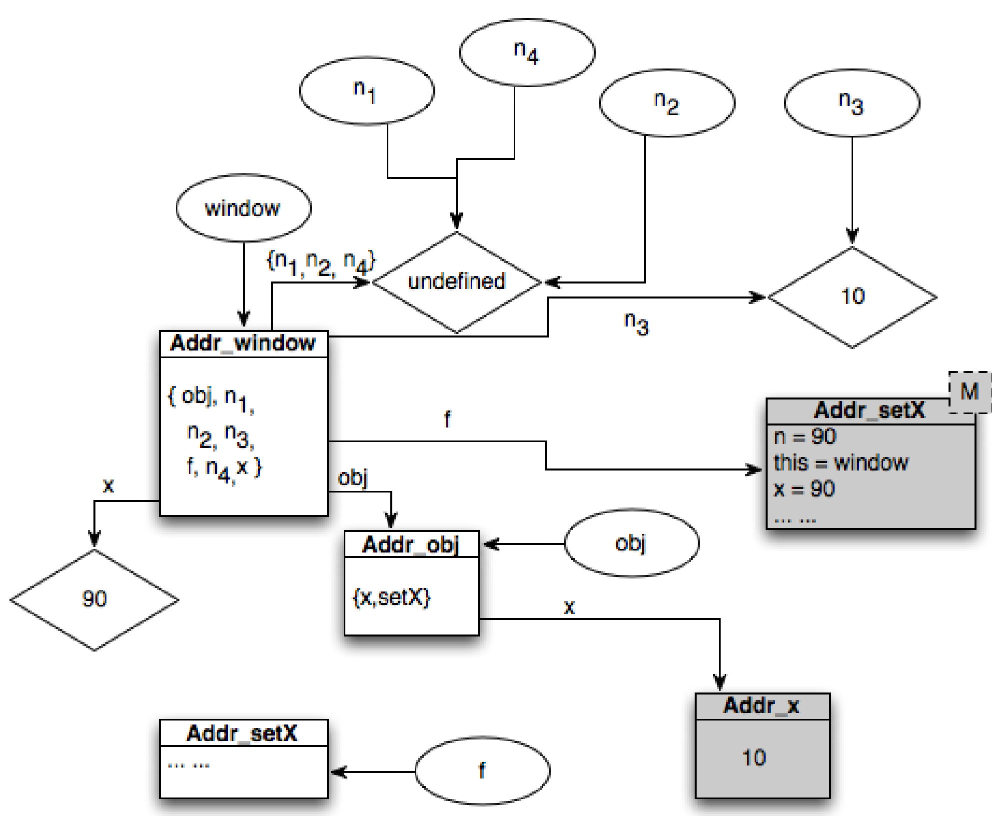

Figure 4.9: Reachability graph after line 14

references edges $\langle$ *Addr_window*, n_5, 10 $\rangle$ and $\langle$ n_5, 10 $\rangle$ are also generated at this point.

Figure 4.11 presents that final reachability graph after line 18. Essentially, it creates an additional label node n_6 pointing to 90, and the reference edge $\langle$ *Addr_window*, n_6, 90 $\rangle$. Thus the return value for the statement $n_6 = window.x$ is 90 rather than 10.

Consider the difference of reachability graph result from Figure 4.8 to Figure 4.9, the **this** variable enclosed in *Addr_setX* multiple object node could be modified from *obj* to *window* after the function invocation $f(90)$ within the global scope. It is clear that full authorisation of the webpage is accessed as long as you are able to manipulate the global object *window*. Therefore, this reachability graph implies that the object n_4 from the allocation site in line 14 directs you to access full authorisation of the underlying webpage.

4.3 Reachability Graph Analysis for $JS^t{}_{sl}$

According to our reachability graph analysis, the relationships among $JS^t{}_{sl}$ program objects can be discovered. We summarise the information in a heap by making a graph. A node represents a heap cell. One node may correspond to possible many nodes. For understanding which node is associated with which node, we view the program as a generator for information, the symbolic execution for each heap allocation statement adds a new node to the graph. At the end of the extraction of program information, a reachability graph is presented to have further analysis. In our reachability graph analysis, program states can be described by reachability graphs according to the executions of program statements (see Figure 4.12). Nodes and reference edges are essential elements in the reachability graph analysis.

Label nodes $ln \in \mathcal{N}_{\mathcal{L}}$ represent the variables in programs. Global primitive nodes $pn \in \mathcal{N}_{\mathcal{P}}$ represent the primitive type value of the variables located in global scope. Heap region nodes $n \in \mathcal{N}_{\mathcal{H}}$ represent the data structures in heap cells. Their properties are

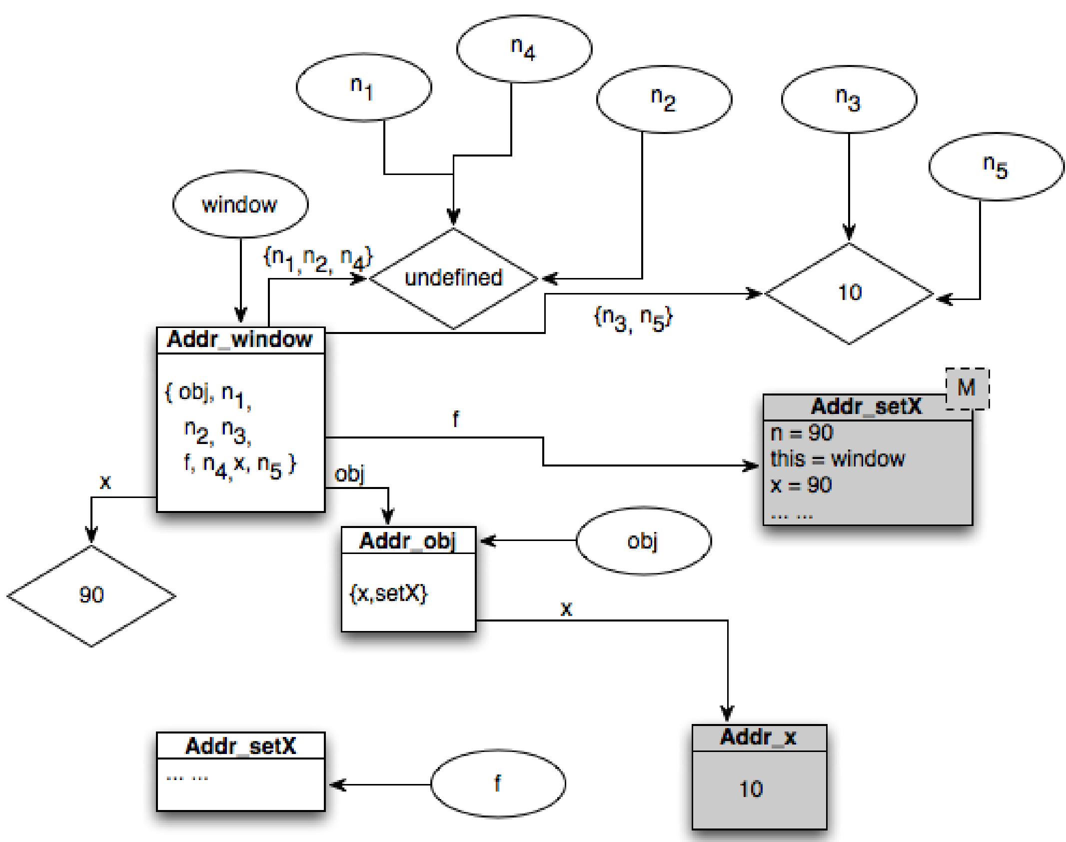

Figure 4.10: Reachability graph after line 16

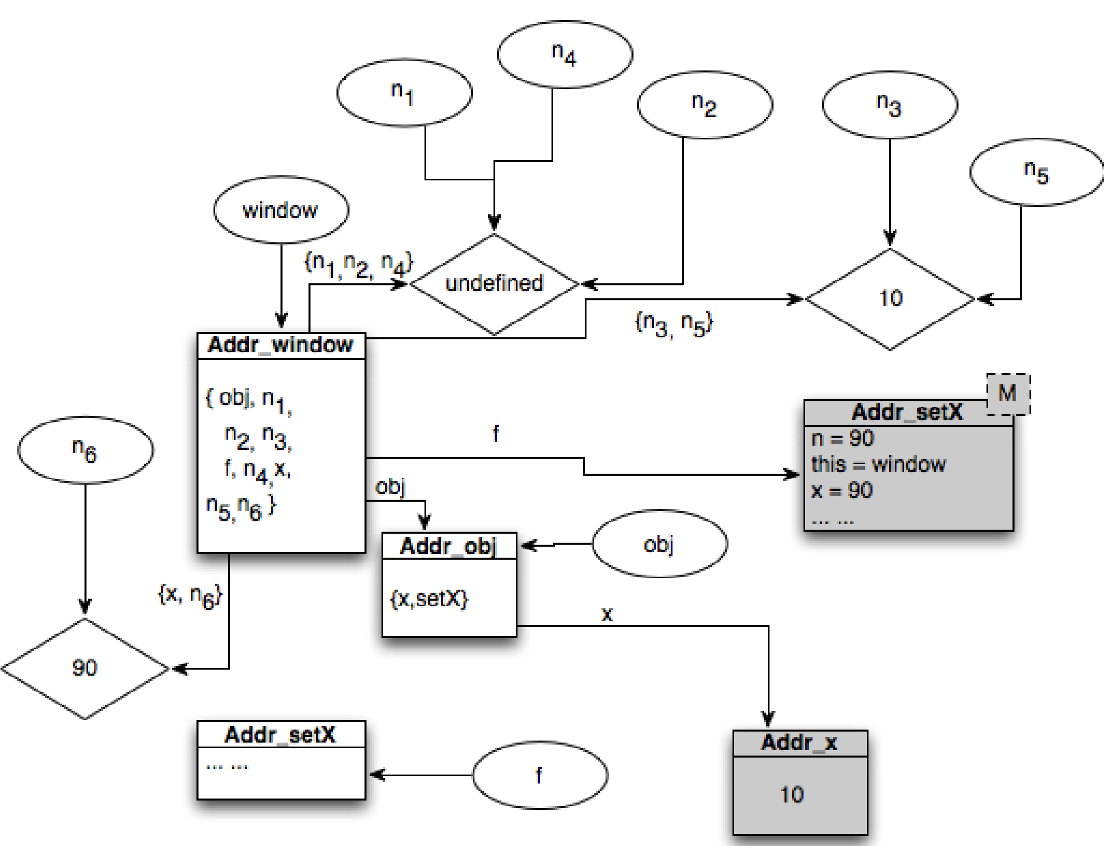

Figure 4.11: Reachability graph after line 18

listed below:

- Heap region nodes can bind a single object, a multiple objects heap region node. or a summary heap region node.
- A heap region node has two parts. One part stores the address of the underlying heap cell. The other part is a *reachability state*, which stores the objects that the given object can reach. For a function or method object, it stores the function body, parameters and **this** variable. Note that, the reachability could not be propagated to involve indirect referenced objects, only the object directly referenced by a reference edge can be reached by the given object.
- A heap region node may have more than one reference edge pointing to or pointing from it.
- We use shade to across a heap region node for heap region node in visualised reachability graphs associated with an allocation site, called summary node. The most recent objects at an allocation site are assigned or updated their heap region node. The older object allocations are firstly copied and updated later in summary nodes.
- We use M label on the top right for multiple objects heap region node when the function is invoked more than one time.

Reference edges $re \in \mathcal{E}$ describe the reachability from one node to another node. Every reference edge between heap region nodes has an associated field $f \in \mathcal{F}$,

$$f \in \mathcal{F} = \textbf{Field}$$

The set of reference edges $\mathcal{E}$ in a reachability graph is defined as follow:

$$\begin{aligned} \mathcal{E} &= \mathcal{N}_{\mathcal{L}} \times \mathcal{N}_{\mathcal{H}} \ \cup \ \mathcal{N}_{\mathcal{L}} \times \mathcal{N}_{\mathcal{P}} \\ &= \cup \, \mathcal{N}_{\mathcal{H}} \times \mathcal{N}_{\mathcal{P}} \ \cup \ \mathcal{N}_{\mathcal{H}} \times \mathcal{N}_{\mathcal{P}} \\ &= \cup \, \mathcal{N}_{\mathcal{H}} \times \mathcal{N}_{\mathcal{H}} \ \cup \ \mathcal{N}_{\mathcal{H}} \times \mathcal{F} \times \mathcal{N}_{\mathcal{H}} \end{aligned}$$

In summary, the reachability graph analysis is to visualise the program state after executing each statement. This analysis is so complex that is not appropriate to be adopted for achieving our aim in this chapter, we only use it for visualising the program status deduced in the final program state.

4.4 Verification

To adapt **this** analysis in our program verification, the abstract syntax of the language is extended from JS_{sl} to $JS^t{}_{sl}$ by employing **this** as a statement. We also modify the specification language mentioned in Chapter 3 from $Spec_{sl}$ to $Spec^t{}_{sl}$. A key characteristic of $Spec^t{}_{sl}$ is that the entire state of the language resides in the object heap. The object heap has various data structures, the preprocessing stage creates an abstract state Δ (see Section 3.5.1) which consists of Π that contains all the global variables referring to heap-insensitive information and Σ that contains the objects referring to heap allocation information, respectively. Through this semantic-preserving program transformation, our reachability graph is able to provide complete information of $JS^t{}_{sl}$ programs. In this section, the syntax of $JS^t{}_{sl}$ program, the extended semantic rules and inference rules are presented. Our aim is to mainly discover the assertions that modifies the value of **this** enclosed in a function to *window* object.

4.4.1 The Language $JS^t{}_{sl}$

The variable **this** is employed as an expression in the syntax of $JS^t{}_{sl}$ (See Figure 4.12). The new statement *FiledMutation* is able to manipulate **this** relevant statement,

$e \in Exp$	$::=$	x	Variable
	$\mid$	v	Primitive Value
	$\mid$	$p(e_1, ..., e_n)$	Arithmetic or Boolean
	$\mid$	$F([x_1, ..., x_n])$	Function
	$\mid$	$this$	this
$ee \in ExpFunc$	$::=$	e	Exp
	$\mid$	$\textbf{func}\ [F]([x_1, ..., x_n])\ \{c\}$	[named] FuncExp
$c \in Statement$	$::=$	$\textbf{skip}$	Skip
	$\mid$	$\textbf{var}\ x = ee$	LocalAssignment
	$\mid$	$x = ee$	GlobalAssignment
	$\mid$	$x = x'.f$	FieldLookup
	$\mid$	$e.f = ee$	FieldMutation
	$\mid$	$\textbf{return}\, e$	Return
	$\mid$	$x = x'.x_0([e_1, ..., e_n])$	FuncCall
	$\mid$	$x = x_0([e_1, ..., e_n])$	FuncCall
	$\mid$	$x = \{f_1 : ee_1, ..., f_n : ee_n\}$	ObjLiteral
	$\mid$	$x = new\, x'()$	ObjCreation
	$\mid$	$x = new\, x'([e_1, .., e_n])$	ObjCrtFunc
	$\mid$	$c;\ c$	Sequencing
	$\mid$	$if\ (e)\ \{c\}\ else\ \{c\}$	Condtional, b is boolean value
	$\mid$	$while\ (e)\ \{c\}$	Iteration
Identifiers			
$F \in FuncID$	$::=$	$F \mid F' \mid ...$	
$f \in FieldID$	$::=$	$f \mid f' \mid f_1 \mid x \mid ...$	
$b \in BooleanID$	$::=$	$True \mid False$	
$x \in VariableID$	$::=$	$x \mid x' \mid x_0 \mid ...$	
$v \in Variable$	$::=$	$int \mid str \mid null \mid undef$	
[...]	$::=$	$optional$	

Figure 4.12: Syntax of ${JS^t}_{sl}$

such as "$this.x_1 = x_2$".

Table 4.1 summarises the different features of the language JS_{sl} and ${JS^t}_{sl}$. In the features row, there are *object structure*, *function invocation*, *prototype inheritance*, *new statement*, *iteration*, *conditional*, *this statement*. The symbol "✔" represents the language supports this feature. The symbol "✘" represents the language does not support this feature.

Feature	JS_{sl}	$JS^t{}_{sl}$
Object Structure	✔	✔
Function Invocation	✔	✔
Prototype Inheritance	✔	✔
new *Statement*	✔	✔
Iteration	✔	✔
Conditional	✔	✔
this variable	✘	✔

Table 4.1: Features Comparison of JS_{sl} and $JS^t{}_{sl}$

As you can see, each of the features from JS_{sl} are "inherited" to $JS^t{}_{sl}$ language. In addition, $JS^t{}_{sl}$ program supports **this** expression and corresponding statements that involve **this**. Therefore, $JS^t{}_{sl}$ has more expressiveness than JS_{sl}.

4.4.2 Revised Operational Semantic Rules

As the language $JS^t{}_{sl}$ supports **this** expression, the operational semantics need to be improved to adapt such feature. According to our previous semantics, the evaluation of the new expression **this** returns either a location of a normal object or a location $\mathsf{loc_w}$ where refers to *window* object. In contrast with the [**op-mutate-field**] rule in chapter 3, the modified rule [**op-mutate-field**] has its premises changed to evaluate the expression e to a nullable location (See Figure 4.13), but he final state after the evaluation of the statement $e.f = ee$ remains the same.

$$\frac{\begin{array}{c} s(e) = \ell \quad \ell \neq null \\ \ell \in dom(h) \quad h(\ell) = r \quad s(ee) = v \end{array}}{e.f = ee, (s, h) \rightarrow (s, h[\ell \mapsto [f \mapsto v]])} \text{ [op-mutated-field]}$$

Figure 4.13: Revised [**op-mutate-field**] Rule

The rest of the operational semantic rules remain exactly the same as in Section 3.4.2, which we omit to explain in this chapter.

4.4.3 Specification Language for $JS^{t}{}_{sl}$

Adding the use of **this** causes a series of chain reaction to other features of the language, such as alias, scope chain, we provide an improved specification language $Spec^{t}{}_{sl}$ (See Figure 4.14) to manage it. Table 4.2 summarises the difference between the language $Spec_{sl}$ and $Spec^{t}{}_{sl}$ from the perspective of expressiveness.

Feature	$Spec_{sl}$	$Spec^{t}{}_{sl}$
Object Structure	✔	✔
Alias	✘	✔
Prototype Chain	✔	✔
Scope Chain	✘	✔
Local Variable	✔	✔
this	✘	✔

Table 4.2: Properties Comparison of $Spec_{sl}$ and $Spec^{t}{}_{sl}$

Applying the language $Spec^{t}{}_{sl}$ in this chapter, we also need to have alteration on the corresponding semantics model that caused by the improvement of the language (See Figure 4.14) particularly with respect to $pure\ formula$ and $heap\ formula$. The different object identifications may point to the same location in the abstract heap h, those identifications are alias to each other, the modification of the location content would cause the change which their identifications map to. To deal with alias analysis, we must identify the objects who refer to the same location. In $Spec^{t}{}_{sl}$ an alias assertion is represented by equality and inequality of two object identifiers, such as $x_1 = x_2$. We join two alias assertion with $\wedge$. Note that, we also consider the case where an object identification is alias with $undefined$ object. Each alias assertion is found as heap-insensitive information. In order to merge with $pure\ formula$, we remain its jointness

Specification	*Specsl*	$::= \{\ \Delta_1\ \}\ \mathbf{c}\ \{\ \Delta_2\ \}$
Abstract state	Δ	$::= \Pi\ \parallel\ \Sigma\ \parallel\ \mathbf{I}$
Pure formula	Π	$::= b \mid \Pi_1 \wedge \Pi_2 \mid \Pi_1 \vee \Pi_2 \mid \exists x.\Pi \mid x \circleddash e \mid a \wedge \Pi$
Heap formula	Σ	$::= \mathbf{emp} \mid x \mapsto r \mid \Sigma_1 * \Sigma_2 \mid \Sigma_1 -\!\!* \Sigma_2 \mid \exists x.\Sigma$
Boolean	b	$::= \mathbf{true} \mid \mathbf{false} \mid x \mid b_1 = b_2$
Variable	x	$::= f \mid x' \mid x_i (i \in Z) \mid \text{this}$
Fields	f	$::= f' \mid f_i (i \in Z)$
Expression	e	$::= F \mid v \mid x \mid const \mid e_i (i \in Z)$
Record	r	$::= [f_1 : e_1, .., f_n : e_n] \mid e$
Alias	a	$::= x_1 = x_2 \mid x_1 \neq x_2 \mid x = undef \mid x \neq undef \mid a_1 \wedge a_2$
Scope Chain	$\mathbf{I}$	$::= LS$

Figure 4.14: The Specification Language $Spec^{t}_{sl}$

with other pure formulas by $\wedge$, such as $a \wedge \Pi$.

At this point, the style of our reasoning is not enough for JS^{t}_{sl}. We must also assert scope chain information when a variable can not be resolved in its current scope. As mentioned in the previous chapters, we have two kinds execution contexts in JS^{t}_{sl} program, including global context and local context. A local context normally refers to a scope constructed by a function statement. $Variable\ Object$ (VO) is a helper object that refers to an execution context (global or local context). A VO contains the information of all the local variables and parameters in the current scope. The locations of variable objects are recorded in the scope chain, we use notation "[ ], LS, $\ell s : LS$, LS(x.@scope)", where "[ ]" is null location, LS is the current scope chain, "$\ell s : LS$" for when location ℓs is in the scope chain LS, LS(x.@scope) for when the location of the object x is attached in the LS by internal property @scope. The root of a scope chain is global scope who has location loc_{go} pointing to object $window$. Each variable object has a pointer @proto pointing to a prototype chain, the end of the prototype chain is object OProto who has

location $\mathsf{loc_{op}}$.

In Figure 4.15, we cater a chain mechanism for the scope and prototype lookup feature. As you can see, the resolving object x from its current scope location ℓs_0 follows the scope chain to the scope ℓs_2 and eventually retrieves its value in the location ℓ along with the prototype chain. Note that in the process of resolving an object x, it is resolved as a field name "x" of the first variable object in the scope chain ℓs_0. Then follows the variable object's prototype chain to its prototype object. After that, the process moves to the next variable object who has location ℓs_1.

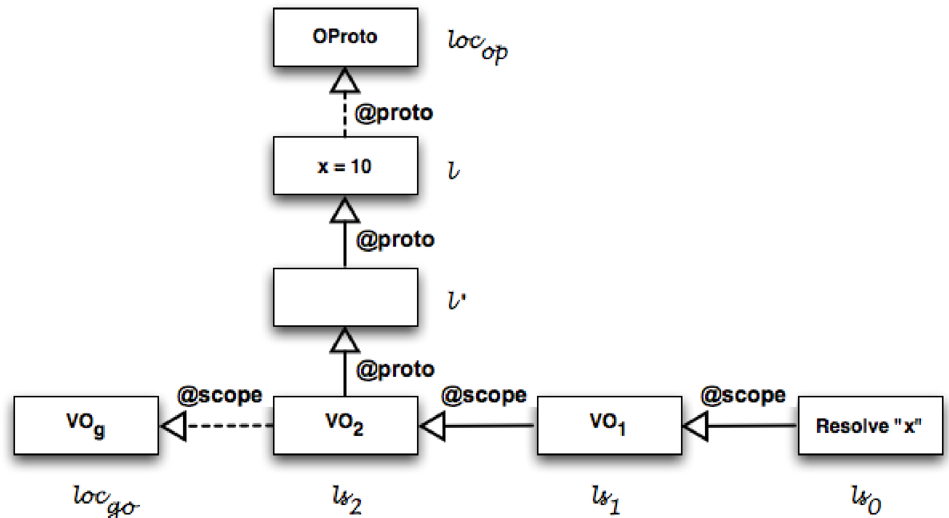

Figure 4.15: Scope and Prototype Lookup Chain

To able to reason this, we extend scope list **I** in *heap formula* due to its heap-sensitive information. The expression **I** is a current list of VOs, in the example of Figure 4.15, the current list of variable objects is given by $\mathrm{I} = [\ell s_0, \ell s_1, \ell s_2, loc_{\mathsf{go}}]$. Note that, we employ LS as the current scope chain, $\ell s \in LS$. We distinguish a scope location ℓs from a normal object location ℓ. The newly created variable object is prepended to the scope list **I**, which means **I** can be used to record the instant scope chain information when it is extended by variable object.

Figure 4.16 presents the semantic model that caters the newly added improvement of the language $Spec^t_{sl}$. Note that, the semantics that does not show in this figure remain the same as defined in $Spec_{sl}$ (See Figure 3.11).

4.4.4 Main Verification Algorithm

The specifications must be subject to a process of refinement before they can actually be implemented. The result of such a refinement process is an executable algorithm. In this section, we formulate our verification algorithm for safe use of **this** variable with the given specification. The algorithm for refinement is given in Algorithm 1 Safe_This algorithm. Line 1 initialises the starting point of a program with given statements, precondition and postcondition information. Line 2 analysis is conducted using a set of symbolic rules to be explained in Section 4.4.6. If the symbolic execution succeeds, the verification moves on to the second step (line 5). However, if the symbolic execution fails at some point, where the current program state cannot satisfy the requirement of the next instruction, the whole verification returns fail at line 15. Along such analysis, if the verification could successfully reach the postcondition Δ_{pos}, otherwise it returns false for Δ_{pos} (line 3), the reachability relationships among objects are derived. Furthermore, the postcondition we might approach is composed of three parts, pure formula, heap formula and scope chain information. Line 6-10 check the heap formula in postcondition

$$
\begin{array}{lll}
s, h \models \Pi \parallel \Sigma \parallel \mathbf{I} & \text{iff} & s, h \models \Pi \parallel \Sigma,\ \mathrm{I} = LS \text{ and } s, h \models LS \\
s, h \models \mathbf{I} & \text{iff} & \mathrm{I} = LS \text{ and } s, h \models LS \\
s \models_A a_1 \bowtie a_2 & \text{iff} & s(a_1) \bowtie s(a_2), \text{where} \bowtie \in \{=, \neq\} \\
s \models_A x \bowtie undef & \text{iff} & s(x) = NaN, \text{where} \bowtie \in \{=, \neq\} \\
s \models_A a_1 \wedge a_2 & \text{iff} & s \models_A a_1, s \models_A a_2 \\
s \models_A a \wedge \Pi & \text{iff} & s \models_A a, s \models_A \Pi
\end{array}
$$

Figure 4.16: The Additional Semantic Model for $Spec^t_{sl}$

regarding **this** variable. Because **this** is a local and internal variable inside of a function. The occurrence of **this** in the postcondition must infer that it also occurs in the precondition. Therefore, if no occurrences of **this** in the precondition and postcondition, it returns safe. The forward analysis deals with the case of **this** occurs in the specification. It returns safe when **this** variable is not able to reach *window* object. If the reachability was established between **this** and *window*, the algorithm returns "not safe".

Definition 3 (Reachibility)**.** *Give a heap formula* $\Sigma = x \mapsto r * \Sigma_1$*, the atomic heap* $x \mapsto r$ *is reachable from a variable* v *if and only if the following recursively defined relation holds:*

$$\begin{aligned} ReachVar(v, \Sigma) \quad &= \quad (ReachVar(v, x \mapsto e) \wedge (x = v)) \vee \\ &\quad\;\; (ReachVar(v, x \mapsto [x_r : e]) \wedge (v = x_r)) \end{aligned}$$

The function **Reach** is used to analyse whether **this** from the postcondition can reach *window* object. It is shown as below:

$$\mathbf{Reach}(\Delta) = \bigcup_{v \in \mathsf{LV}(\Delta)} \mathbf{ReachVar}(v, \Delta) \ \textit{where} \ \Delta ::= \Pi \parallel \Sigma \parallel \mathbf{I}$$

This $\mathbf{Reach}(\Delta)$ function returns a set of variables that are reachable from the free variables in the abstract state Δ.

The function $\mathbf{ReachVar}(v, \Pi \parallel \Sigma \parallel \mathbf{I})$ returns the minimal set of variables which satisfies the following relation:

$$\{v\} \cup \{x_2 \mid \exists x_1, \Pi_1 \cdot x_1 \in \textbf{ReachVar}(v, \Pi \parallel \Sigma \parallel \mathbf{I}) \wedge \Pi = (x_2 = x_1 \wedge \Pi_1)\} \cup$$
$$\{x_2 \mid \exists x_1, \Sigma_1 \cdot x_1 \in \textbf{ReachVar}(v, \Pi \parallel \Sigma \parallel \mathbf{I}) \wedge \Sigma = (x_1 \mapsto [x_2 : e] * \Sigma_1)\}$$
$$\subseteq \textbf{ReachVar}(v, \Delta)$$

Algorithm 1 Safe_This Algorithm

```
1:  procedure SAFE_THIS(c, Δ_pre, Δ_pos)
2:      if {Δ_pre}c{Δ_pos} then
3:          if Δ_pos = false then return fail
4:          else
5:              Δ_pos := Π_pos ∥ Σ_pos ∥ I
6:              if this ∉ LV(Σ_pos) then return SAFE
7:              else
8:                  if window ∉ Reach(Δ_pos) then return SAFE
9:                  else
10:                     return NOT SAFE
11:                 end if
12:             end if
13:         end if
14:     else
15:         return fail
16:     end if
17: end procedure
```

4.4.5 Formal Property: Safety

According to our reachability graph analysis, we are able to produce effective reachability predicates for describing data structures on the graphs of its revolution. To prevent the malicious use of **this** that could be implicitly modified to bind to the global object *window*, it reduced to find the reachability predicates that indicate the value of **this** variable within a function or method is unexpectedly changed to point to *window*. Thus we have safety definition for ${JS^t}_{sl}$ program:

Definition 4 (Safety-1)**.**

A statement c is safe, if the relationship between the **this** *variable in a given abstract state* Δ_{pre} *and its transited variable* $this'$ *in the abstract state* Δ_{pos} *can be detected. That is,* $this' \in \mathsf{LV}(\Sigma_{pos})$*. And the variable that are reachable from the free variable in the state* Σ_{pos} *does not include the window object. That is,* $window \notin \textbf{\textit{Reach}}(\Delta_{pos})$*.*

Note that, our $safety$ definition only consider the case when the activated statement involves operations on functions. Because **this** safety issue occurs in the context of functions. In the case of no **this** variable employed statement, such as postconditions do not own **this** variable, it always returns safe.

Definition 5 (Safety-2)**.** *For all* $JS^t{}_{sl}$ *statements c, if* **this** *is safe in* c_i *(*$i \in \mathbb{Z}$*), and* $c_i \twoheadrightarrow c_{i+1}$*, then* **this** *is safe in* c_{i+1}*.*

The definition 4 shows the transitivity of $safety$. If the current activated statement c_i is safe, and $c_i \twoheadrightarrow c_{i+1}$ indicates that statements move from the i_{th} to $(i+1)_{th}$. The expression $\twoheadrightarrow$ represents the movement of a sequence of statements. For example, if we have a function statement $x = func(n)\{c\}$ as a field of a literal object obj, the invocation $obj.x(n)$ has been proved safe. At this point, **this** enclosed in $obj.x(n)$ statement is also safe in the next statement, except when the coming up statement is interfering the value of **this** in the function statement $x = func(n)\{c\}$, such as calling the function by a global variable f, such as $f(n)$ where $f = obj.x$. As it may be observed, **this** is considered to be safe or unsafe with respect to its enclosed statement. In above example, **this** is safe enclosed in the statement $obj.x(n)$, but not safe enclosed in the statement $f(n)$.

Theorem 2. A $JS^t{}_{sl}$ program is safe when its all statements are safe.

Our revised axiomatic framework in this chapter employs **this**, but we must prove theorem 2. It is sufficient to prove the following lemmas: [1]

Lemma 1 (Safety)**.** *For a* $JS^t{}_{sl}$ *statement* c_i*, if the* Algorithm 1 Safe_This algorithm *returns SAFE, then* c_i *is safe.*

[1] Addition proof details are in the Appendix.

Lemma 2 (Subject Reduction)**.** *For a $JS^t{}_{sl}$ statement c_i, if c_i is safe, then c_{i+1} is safe.*

Those lemmas are proved by the induction of inference rules (see Section: 4.4.6). However, we have to ensure that our revised semantic rules do not violate the previous semantic model in chapter 3 and safety (lemma 1).

4.4.6 Revised Inference Rules

For our reasoning rules, we employ α logic predicate to present the reasoning about scope chain. To consider resolving a variable, walk down the *variable object* from the scope chain when searching for a variable x can not be found in its current scope. What is of interest to us is the order in which *variable objects* will be checked. Notice that the variable x will be searched following by @proto pointer when it is recognised in a *variable object* location. The α predicate is to precisely capture the VOs that must be checked.

In Figure 4.17, we define the semantics for α predicate by satisfaction relation. The predicate $\alpha([LLS], \ell : LS, x, \ell)$ holds only for abstract heap h such that the variable x can be resolved in the *variable object* location (or a prototype of the object) at location ℓ. The first argument $[LLS]$ in our predicate "precisely" specifies the concrete locations in heap cells which must be visited, it determine the tracks of scope list. For example, recall the illustration in Figure 4.15, if the prototype of the heap cell ℓs_2 has location ℓ', the predicate $\alpha([[\ell s_0], [\ell s_1], [\ell s_2, \ell', \ell]], \mathbf{I}, x, \ell s_2)$ is satisfied by the abstract heap h possessing the variable x and internal property @proto for the objects who have location ℓ', ℓ. Note that it is not necessary to walk down every VO, the visit stops at the point of discovering location of the variable x.

We update the inference rules to deal with scope chain, and **this** features expended in $JS^t{}_{sl}$ regarding safety property reasoning. The assertion language of the $JS^t{}_{sl}$ remains the same as in Section 3.5.1. Figure 4.18, Figure 4.19, Figure 4.20, Figure 4.21 present the revised inference rules. In those rules, the pure formula and heap formula of the

program abstract state remain the same as mentioned in Chapter3 except the scope chain information **I**. The rules we mainly explain are the ones who has effect on the revision of the current scope chain information.

In Figure 4.18, we explain rule [**sl-glob-assign2**], after a function object is globally created, the Variable Object of this function object is attached at a specific location in the scope chain list. The postcondition formula $LS(x.\text{@scope})$ adds location (ℓs_x) into LS by linking through @scope internal property. For rule [**sl-local-assign2**], the precondition has OV location ℓs_{x_0} is the scope chain LS. For example the function object x is declared inside of the function x_0, the ℓs_{x_0} has already existed in the scope chain LS. After the execution of the location function assignment statement, the postcondition adds the OV location (ℓs_x) into LS, the scope chain is updated from $\mathbf{I}$ to $\mathbf{I}'$. For the rest of the rules in Figure 4.18, the execution of their statements do not alter the status of the scope chain.

In Figure 4.19, the execution of their statements do not alter the status of the scope chain because that a function need to be defined first before its invocation. Thus in each precondition, we have the formula $\mathbf{I} \doteq \ell s_{x_0} : LS$ that is OV location of the function object x_0 in the scope chain. The postcondition remains the same scope chain information as in the precondition.

In Figure 4.20, the precondition of the rule [**sl-obj-crt-fun**] has the OV location $\ell s'_x$ in the scope chain LS, its postcondition adds a new OV location (ℓs_x) into L by the formula $\mathbf{I}' \doteq (x.\text{@scope})$.

$$
\begin{array}{lll}
s,h \models \alpha([\,],[\,],_,null) & \text{iff} & dom(h)=\emptyset \\
s,h \models \alpha([LLS],\ell:LS,x,\ell) & \text{iff} & \exists h_1,h_2,\ell,v\cdot h_1\#h_2 \text{ and } h=h_1*h_2 \\
& & \quad \text{and } \ell\in dom(h_1) \text{ and } h_1(\ell)(x)=v \\
& & \quad \text{and } \ell\in dom(h_2) \text{ and } \ell:\ell s \text{ and } \ell s : LS
\end{array}
$$

Figure 4.17: Logic Predicate Semantic Model for ${Spec^t}_{sl}$

$$\frac{}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}\mathbf{skip}\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}}$$ [sl-skip]

$$\frac{}{\{\Pi[e/x] \parallel \Sigma \parallel \mathbf{I} \doteq LS\}x = e\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}}$$ [sl-glob-assign1]

$$\frac{}{\{\Pi \parallel \Sigma[e/x] \parallel \mathbf{I} \doteq LS\}\mathbf{var}\ x = e\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}}$$ [sl-local-assign1]

$$\frac{r = [\mathbf{body} : c, \mathbf{params} : (x_1, ..., x_n), \text{@proto} : \mathsf{loc_{op}}]}{\{\Pi \parallel \mathbf{emp} \parallel \mathbf{I} \doteq LS\}x = \mathbf{func}[F](x_1, ..., x_n)\{c\}\{\Pi \parallel \exists x \cdot x \mapsto r \parallel \mathbf{I}' \doteq LS(x.\text{@scope})\}}$$ [sl-glob-assign2]

$$\frac{\Sigma \equiv \Sigma_0 * x_0 \mapsto r_0 \quad x_0 \in \mathbf{Func} \\ r = [\mathbf{body} : c, \mathbf{params} : (x_1, ..., x_n), \text{@proto} : \mathsf{loc_{op}}] \\ x \notin \mathsf{LV}(\Sigma) \quad \mathbf{I} \doteq \ell s_{x_0}{:}LS}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}\mathbf{var}\ x = \mathbf{func}[F](x_1, ..., x_n)\{c\}\{\Pi \parallel \Sigma * (\exists x \cdot x \mapsto r) \parallel \mathbf{I}' \doteq LS(x.\text{@scope})\}}$$ [sl-local-assign2]

$$\frac{r = [..., f : v, ...]}{\{\Pi \parallel x' \mapsto r \parallel \mathbf{I} \doteq LS\}x = x'.f\{(\exists x \cdot \Pi) \wedge x{=}v \parallel x' \mapsto r \parallel \mathbf{I} \doteq LS\}}$$ [sl-lookup-field]

$$\frac{x' \notin \mathsf{LV}(\Sigma) \quad f \notin \mathsf{dom}(r) \\ r(\text{@proto}) = x'' \quad \{\Pi \parallel \Sigma\}x = x''.f\{\Pi' \parallel \Sigma'\}}{\{\Pi \parallel x' \mapsto r * \Sigma \parallel \mathbf{I} \doteq LS\}x = x'.f\{\Pi' \parallel x' \mapsto r * \Sigma' \parallel \mathbf{I} \doteq LS\}}$$ [sl-lookup-proto]

$$\frac{\Sigma \equiv x' \mapsto \mathsf{OProto} \\ f \notin \mathsf{LV}(\Sigma)}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}x = x'.f\{\exists x \cdot \Pi \wedge x{=}\mathsf{undef} \parallel \Sigma \parallel \mathbf{I} \doteq LS\}}$$ [sl-lookup-undef]

$$\frac{r = [..., f : v, ...] \ \ \textbf{or else} \ \ f \notin \mathsf{dom}(r)}{\{\Pi \parallel x \mapsto r \parallel \mathbf{I} \doteq LS\}x.f = ee\{\Pi \parallel x \mapsto r[f \mapsto ee] \parallel \mathbf{I} \doteq LS\}}$$ [sl-mutate-field]

Figure 4.18: Updated Inference Rules for Variable Assignments and Field Statements

$$\frac{\begin{array}{c}\Sigma \equiv \Sigma_0 * x'{\mapsto}[x_0{:}x'', ..] * x''{\mapsto}[\mathbf{body} : c, \mathbf{params} : (x_1..x_n), ...] \\ \Sigma_1 \equiv \Sigma * x_0 \mapsto [\text{this: } x', x_1 : e_1, ..., x_n : e_n, ...] \\ \{\Pi \parallel \Sigma_1\}c\{\Pi_1 \parallel \Sigma_2\} \quad \mathbf{I} \doteq \ell s_{x_0}{:}LS\end{array}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}x = x'.x_0([e_1, ..., e_n])\{(\exists x \cdot \Pi_1) \wedge x = \mathbf{res} \parallel \Sigma_2 \parallel \mathbf{I}' \doteq LS\}} \quad \text{[sl-fun-call-obj]}$$

$$\frac{\begin{array}{c}x' \notin \mathsf{LV}(\Sigma) \quad x_0 \notin \mathsf{dom}(r) \quad r(\text{@}\mathsf{proto}) = x'' \quad \mathbf{I} \doteq \ell s_{x_0}{:}LS \\ \{\Pi \parallel \Sigma\}x = x''.x_0([e_1, ..., e_n])\{\Pi' \parallel \Sigma'\}\end{array}}{\{\Pi \parallel x' \mapsto r * \Sigma \parallel \mathbf{I} \doteq LS\}x = x'.x_0([e_1, ..., e_n])\{\Pi' \parallel \Sigma' \parallel \mathbf{I}' \doteq LS\}} \quad \text{[sl-fun-call-proto]}$$

$$\frac{\begin{array}{c}\Sigma \equiv x' \mapsto \mathsf{OProto} \\ x_1 \notin \mathsf{LV}(\Sigma) \quad \mathbf{I} \doteq \ell s_{x_0}{:}LS\end{array}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}x = x'.x_0([e_1, ..., e_n])\{(\exists x \cdot \Pi) \wedge x{=}\mathsf{undef} \parallel \Sigma \parallel \mathbf{I} \doteq LS\}} \quad \text{[sl-fun-undef]}$$

$$\frac{\begin{array}{c}\Sigma \equiv (\Sigma_1 * x_0 \mapsto [\mathbf{body} : c, \mathbf{params} : (x_1...x_n), \text{@}\mathsf{proto}{:}\mathsf{loc_{fp}}, ...]) \\ \Sigma_1 \equiv (\Sigma * x' \mapsto [\text{this: } \mathsf{loc_w}, x_1 : e_1, ..., x_n : e_n, ...]) \\ \{\Pi \parallel \Sigma_1 \parallel \mathbf{I}\}c\{\Pi_1 \parallel \Sigma_2 \parallel \mathbf{I}_1\} \quad \mathbf{I} \doteq \ell s_{x_0}{:}LS\end{array}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}x = x_0([e_1, ..., e_n])\{(\exists x \cdot \Pi_1) \wedge x = \mathbf{res} \parallel \Sigma_2 \parallel \mathbf{I}_2 \doteq LS\}} \quad \text{[sl-fun-call-dir]}$$

$$\frac{}{\{\Pi[e/\mathbf{res}] \parallel \Sigma \parallel \mathbf{I}\}\mathbf{return}\ e\{\Pi \parallel \Sigma \parallel \mathbf{I}\}} \quad \text{[sl-return]}$$

Figure 4.19: Updated Inference Rules for Function Invocation

$$\frac{r = [f_1 : e_1, ..., f_n : e_n]}{\{\Pi \parallel \mathbf{emp} \parallel \mathbf{I} \doteq LS\}\ x = \{f_1 : e_1, .., f_n : e_n\}\ \{\Pi \parallel \exists x \cdot x \mapsto r \parallel \mathbf{I} \doteq LS\}}\quad \textbf{[sl-obj-crt-literal]}$$

$$\frac{}{\{\Pi \parallel x' \mapsto r \parallel \mathbf{I} \doteq LS\}\ x = new\, x'()\ \{\Pi \parallel x' \mapsto r * (\exists x \cdot x \mapsto [\text{@proto} : x']) \parallel \mathbf{I} \doteq LS\}}\quad \textbf{[sl-obj-crt-new]}$$

$$\frac{\begin{array}{c}\Sigma \equiv (\Sigma_0 * x' \mapsto [\mathbf{body} : c, \mathbf{params} : (x_1..x_n), \text{@proto:loc}_{\text{op}}, \text{this:loc}_{\text{w}}]) \\ \Sigma_1 \equiv \Sigma * (\exists x'' \cdot x'' \mapsto [\text{this: } x, x_1 : e_1, ...x_n : e_n, ...]) \\ \{\Pi \parallel \Sigma_1\}c\{\Pi_1 \parallel \Sigma_2\} \quad \mathbf{I} \doteq \ell s_{x'}{:}LS\end{array}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}x = new\, x'([e_1, ..., e_n])\{\Pi_1 \parallel \Sigma_2 \parallel \mathbf{I}' \doteq LS(x.\text{@scope})\}}\quad \textbf{[sl-obj-crt-fun]}$$

Figure 4.20: Updated Inference Rules for Object Creation

$$\frac{\begin{array}{c}\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}c_1\{\Pi_1 \parallel \Sigma_1 \parallel \mathbf{I}_1 \doteq LS\} \\ \{\Pi_1 \parallel \Sigma_1 \parallel \mathbf{I}_1 \doteq LS\}c_2\{\Pi_2 \parallel \Sigma_2 \parallel \mathbf{I}_2 \doteq LS\}\end{array}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}c_1;\ c_2\{\Pi_2 \parallel \Sigma_2 \parallel \mathbf{I}_2 \doteq LS\}}\quad \textbf{[sl-sequential]}$$

$$\frac{\begin{array}{c}\{\Pi \wedge b \parallel \Sigma \parallel \mathbf{I} \doteq LS\}c_1\{\Pi_2 \parallel \Sigma_2 \parallel \mathbf{I}_2 \doteq LS\} \\ \{\Pi \wedge \neg b \parallel \Sigma \parallel \mathbf{I} \doteq LS\}c_2\{\Pi_2 \parallel \Sigma_2 \parallel \mathbf{I}_2 \doteq LS\}\end{array}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}\mathbf{if}\ (b)\{c_1\}\mathbf{else}\ \{c_2\}\{\Pi_2 \parallel \Sigma_2 \parallel \mathbf{I}_2 \doteq LS\}}\quad \textbf{[sl-conditional]}$$

$$\frac{\{\Pi \wedge b \parallel \Sigma \parallel \mathbf{I} \doteq LS\}c\{\Pi \parallel \Sigma \parallel \mathbf{I}' \doteq LS\}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}\mathbf{while}\ (b)\{c\}\{\Pi \wedge \neg b \parallel \Sigma \parallel \mathbf{I}' \doteq LS\}}\quad \textbf{[sl-iteration]}$$

Figure 4.21: Updated Inference Rules for Control Structures

4.4.7 Soundness

Based on the definition above we have a theorem for our axiomatic framework in this chapter:

Theorem 3. *Our axiomatic framework presented in Chapter 4 is sound with respect to the underlying operational semantics.*

The soundness proof of the axiomatic frameworks presented in this chapter are in the appendixes B on page 158.

4.5 Summary

It is both practical and challenging problem to verify both function correctness and safety of heap-manipulating ${JS^t}_{sl}$ program. In this chapter, we provide an axiomatic framework to solving it by inferring expected specification mainly for **this** variable related statement from their calling contexts. The framework is proven sound and the ${JS^t}_{sl}$ program is proven correct on condition that the function definition and invocation statement meet the inferred specifications. We employ a forward program reachability analysis over the inferred specification to synthesise the specifications of the **this** safe. In our next chapter, we evaluate the viability of the proposed approach.

Chapter 5

Case Studies and Evaluation

A case study is one which investigates the scenario that can only be studied or understood in context to answer specific research questions and which seeks a range of different kinds of evidence, evidence which is there in the case setting, and which has to be abstracted and collated to get the best possible answers to the research questions – Bill Gillham (Computer scientist in Glasgow University)

5.1 Introduction

This chapter contains two parts, the part one presents the case studies of ${JS^t}_{sl}$ verification framework. Part two is the evaluation of the framework by a comparison work between the ${JS^t}_{sl}$ framework and other frameworks.

5.2 Case Studies

There are four case studies used to evaluate the ${JS^t}_{sl}$ framework in terms of verifying functional correctness and safety. Case study A is employed to show that the framework

is capable of processing the flexible and complex features of JavaScript language by using the example presented in Chapter 3. Case study B and C are used to show that the framework can detect unsafe JavaScript applications before causing catastrophic problems. The case study D shows the scenario that cause the $JS^t{}_{sl}$ framework to be failed.

Each case study is presented in four phases:

Phase 1: Program translation

This phase shows the translation from a JavaScript program to a $JS^t{}_{sl}$ program in a semantic preserving approach. Each expression and statement of JavaScript program is rewritten in $JS^t{}_{sl}$ language for further analysis. As matter as fact, the translation of a JavaScript program to $JS^t{}_{sl}$ can be fully automated by using syntax tree generators. Firstly, a JavaScript program can be translated into a syntax tree, which represents an abstract syntactic structure of source code of the program. The $JS^t{}_{sl}$ program is generated automatically by the syntax tree.

Phase 2: Program analysis

This phase contains two parts. Part one presents the given program specifications from users which are requested to be proved functional correctness. The specifications are expressed in the form of $\{P\}C\{Q\}$, C is program statement, P and Q are assertions written in $Spec^t{}_{sl}$ language.

Part two reveals how the inference rules can be applied to automatically analyse a $JS^t{}_{sl}$ program. The analysis begins from an initial precondition assertion Δ_{pre}, it has value $\{true \parallel emp \parallel \mathbf{I}\}$ where the $pure\ formula$ is true, the $heap\ formula$ is empty, and the predicate for describing scope chains is $\mathbf{I}$. It produces a postcondition assertion Δ_{pos} that specifies the final status of program.

Therefore, if the postcondition assertions match the requested specification from users, it shows that functional correctness of programs is proved, otherwise the

system fails to verify functional correctness.

Phase 3: Algorithm application

In this phase, the postcondition assertion Δ_{pos} produced in the phase 2 will be used as a parameter in the execution of "Safe_ This" algorithm that is defined in Section 4.4.4. The output of this phase produces the detection result of safety for programs.

Phase 4: Result

In this phase, the result of applying verification framework $JS^t{}_{sl}$ will be directly shown to users whether or not the given program is safe.

5.2.1 Case Study A

The case study A employs the JavaScript program example that is presented in Chapter 3, Section 3.3 (see Figure 5.1). In this example, it shows the flexible features of JavaScript including object literal, nested function, prototype inheritance, and dynamic features including the generation of an object field on the fly. The analysis should indicate that this is a safe program. More detailed description of this example is shown in Section 3.3 .

Phase 1: Program translation

The input of this phase is the JavaScript program showed in Figure 5.1. The output is a semantically equivalent $JS^t{}_{sl}$ program shown in Figure 5.2.

Phase 2: Program Analysis

According to the specification language $Spec^t{}_{sl}$ defined in Section 4.4.3, the given

```
1  <script type="text/javascript">
2  var obj = {
3     f1: 1,
4     f2: function(n) {
5        var g = function() {
6           if (n >= 10) { return 2;}
7           else { return 3;}
8        }
9        return g();
10    }
11 };
12 alert(obj.f2(11));        //2
13 alert(obj.f3);            //undefined
14 obj.f3 = 5;
15 alert(obj.f3);            //5
16 res = obj.f2(1);          //3
17 alert(res);
18 </script>
```

Figure 5.1: JavaScript for Case Study A

```
1  obj = {
2     f1: 1,
3     f2: func(n) {
4        var g = func() {
5           if (ge(n,10)) { return 2 }
6           else { return 3 }
7        }
8        var x = g();
9        return x
10    }
11 };
12 n1 = obj.f2(11);
13 n2 = obj.f3;
14 obj.f3 = 5;
15 n3 = obj.f3
16 res = obj.f2(1)
```

Figure 5.2: ${JS^t}_{sl}$ for Case Study A (Phase 1)

specifications that are requested to be proved are given as:

$$\{true \parallel emp \parallel \mathbf{I}\}$$
$$C$$
$$\{n1 = 2 \wedge n2 = \mathsf{undef} \wedge n3 = 5 \wedge \text{res} = 3 \parallel \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}, \text{f3} : 5] \parallel \mathbf{I}'\}$$

where C is a sequence of ${JS^t}_{sl}$ statements.

There are several definitions defined in this phase for assisting the proof in the later phases. These definitions are:

- O_{f2} represents the function object referred to by f2.

$$\begin{aligned} O_{\text{f2}} = [&body : \{var\ \text{g} = func()\{if\ (ge(n, 10))\{return\ 2;\ \} \\ &\qquad\qquad else\ \{return\ 3;\ \}\}; \\ &\quad var\ \text{x} = \text{g}();\ return\ \text{x}\}, \\ ¶ms : (n), @\mathsf{proto} : \mathsf{OProto}] \end{aligned}$$

- O_{g} represents the function object referred to g.

$$\begin{aligned} O_{\text{g}} = [&body : if(ge(n, 10))\ \{return\ 2;\ \}else\ \{return\ 3;\ \} \\ ¶ms : (), @\mathsf{proto} : \mathsf{OProto}] \end{aligned}$$

Figure 5.3 shows that the analysis process is in the form of "{precondition} C {postcondition}". The inference rules that are applied are [**sl-obj-crt-literal**], [**sl-fun-call-obj**], [**sl-lookup-undef**],[**sl-mutate-filed**], [**sl-lookup-field**], and [**sl-fun-call-obj**]. A summary of output in this phase is a final postcondition assertion and is given as:

$\Delta_{pos} = \{\exists n1, n2, n3, res \cdot true \wedge n1 = 2 \wedge n2 = \mathsf{undef} \wedge n3 = 5 \wedge res = 3 \parallel$
$\exists \text{obj} \cdot \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}, \text{f3} : 5] * f2 \mapsto [\mathbf{this} : obj, x1 : 1, \ldots] \parallel \mathbf{I}' \doteq$

$\{true \parallel emp \parallel \mathbf{I}\}$

obj = {f1 : 1, f2 : ...}; **[sl-obj-crt-literal]**

$\{true \parallel \exists \text{obj} \cdot \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}] \parallel \mathbf{I}\}$

n1 = $obj.f2(11)$; **[sl-fun-call-obj]**

$\{true \parallel \exists \text{obj} \cdot \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}] * f2 \mapsto [\mathbf{this} : obj, x1 : 11, \ldots] \parallel \mathbf{I}\}$

var g = $func()\{...\}$;

$\{true \parallel \exists \text{obj} \cdot \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}] * f2 \mapsto [\mathbf{this} : obj, x1 : 11, \ldots] * g \mapsto O_{\text{g}} \parallel \mathbf{I}' \doteq LS(g.\text{@scope})\}$

var x = g();

$\{true \parallel \exists \text{obj} \cdot \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}] * f2 \mapsto [\mathbf{this} : obj, x1 : 11, \ldots] * g \mapsto O_{\text{g}} \parallel \mathbf{I}' \doteq LS(g.\text{@scope})\}$ **[sl-glob-assign2]**

[sl-fun-call-dir]

$if(ge(n, 10))\{return\ 2\}else\ \{return\ 3\}$; **[sl-conditional]**

$\{\exists x \cdot true \wedge x = 2 \parallel \exists \text{obj} \cdot \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}] * f2 \mapsto [\mathbf{this} : obj, x1 : 11, \ldots] * g \mapsto O_{\text{g}} \parallel \mathbf{I}' \doteq LS(g.\text{@scope})\}$

$return\ x$;

$\{\exists n1 \cdot true \wedge n1 = 2 \parallel \exists \text{obj} \cdot \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}] * f2 \mapsto [\mathbf{this} : obj, x1 : 11, \ldots] * g \mapsto O_{\text{g}} \parallel \mathbf{I}' \doteq LS(g.\text{@scope})\}$

$\{\exists n1 \cdot true \wedge n1 = 2 \parallel \exists \text{obj} \cdot \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}] * f2 \mapsto [\mathbf{this} : obj, x1 : 11, \ldots] * g \mapsto O_{\text{g}} \parallel \mathbf{I}' \doteq LS(g.\text{@scope})\}$

n2 = $obj.f3$; **[sl-lookup-undef]**

$\{\exists n1, n2 \cdot true \wedge n1 = 2 \wedge n2 = \text{undef} \parallel \exists \text{obj} \cdot \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}] * f2 \mapsto [\mathbf{this} : obj, x1 : 11, \ldots] \parallel \mathbf{I}' \doteq LS(g.\text{@scope})\}$

obj.f3 = 5; **[sl-mutate-field]**

$\{\exists n1, n2 \cdot true \wedge n1 = 2 \wedge n2 = \text{undef} \parallel \exists \text{obj} \cdot \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}, \text{f3} : 5] * f2 \mapsto [\mathbf{this} : obj, x1 : 11, \ldots] \parallel \mathbf{I}' \doteq LS(g.\text{@scope})\}$

n3 = $obj.f3$; **[sl-lookup-field]**

$\{\exists n1, n2, n3 \cdot true \wedge n1 = 2 \wedge n2 = \text{undef} \wedge n3 = 5 \parallel \exists \text{obj} \cdot \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}, \text{f3} : 5] * f2 \mapsto [\mathbf{this} : obj, x1 : 11, \ldots] \parallel \mathbf{I}' \doteq LS(g.\text{@scope})\}$

res = obj.f2(1) **[sl-fun-call-obj]**

$\{\exists n1, n2, n3, res \cdot true \wedge n1 = 2 \wedge n2 = \text{undef} \wedge n3 = 5 \wedge res = 3 \parallel \exists \text{obj} \cdot \text{obj} \mapsto [\text{f1} : 1, \text{f2} : O_{\text{f2}}, \text{f3} : 5] * f2 \mapsto [\mathbf{this} : obj, x1 : 1, \ldots] \parallel \mathbf{I}' \doteq LS(g.\text{@scope})\}$

Figure 5.3: Program Analysis for Case Study A (Phase 2)

$LS(g.@\mathsf{scope})\}$

As you can see, Δ_{pos} not only matches the given specifications requested to be proved, but also provides more information about programs, which is $f2 \mapsto [\mathbf{this} : obj, x1 : 11, \ldots]$ located in the heap. Therefore, the analysis shows that the programs are functional correct.

Phase 3: Algorithm Application

The algorithm SAFE_THIS($C, \Delta_{pre}, \Delta_{pos}$) has three different inputs, C, Δ_{pre}, and Δ_{pos}. The parameter C is the given program as shown in Figure 5.2. The parameter Δ_{pre} and Δ_{pos} are:

$$\Delta_{pre} = \{true \parallel emp \parallel \mathbf{I}\}$$

$$\Delta_{pos} = \{\exists n1, n2, n3, res \cdot true \wedge n1 = 2 \wedge n2 = \mathsf{undef} \wedge n3 = 5 \wedge res = 3 \parallel \exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{f1} : 1, \mathrm{f2} : O_{\mathrm{f2}}, \mathrm{f3} : 5] * f2 \mapsto [\mathbf{this} : obj, x1 : 1, \ldots] \parallel \mathbf{I}' \doteq LS(g.\mathsf{@scope})\}$$

procedure $Execu_SAFE_THIS$(c,Δ_{pre}, Δ_{pos})

if $\{\Delta_{pre}\}$ c $\{\Delta_{pos}\}$ **then** $[Algorithm1 - line2]$

> **if** $\{true \parallel emp \parallel \mathbf{I}\} c \{\exists n1, n2, n3, res \cdot true \wedge n1 = 2 \wedge n2 = \mathsf{undef} \wedge n3 = 5 \wedge res = 3 \parallel \exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{f1} : 1, \mathrm{f2} : O_{\mathrm{f2}}, \mathrm{f3} : 5] * f2 \mapsto [\mathbf{this} : obj, x1 : 1, \ldots] \parallel \mathbf{I}' \doteq LS(g.\mathsf{@scope})\}$ **then**

$\Delta_{pos} := \Pi_{pos} \parallel \Sigma_{pos} \parallel \mathbf{I}$ $[Algorithm1 - line5]$

> $\Delta_{pos} = \{ \exists n1, n2, n3, res \cdot true \wedge n1 = 2 \wedge n2 = \mathsf{undef} \wedge n3 = 5 \wedge res = 3 \parallel \exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{f1} : 1, \mathrm{f2} : O_{\mathrm{f2}}, \mathrm{f3} : 5] * f2 \mapsto [\mathbf{this} : obj, x1 : 1, \ldots] \parallel \mathbf{I}' \doteq LS(g.\mathsf{@scope})\}$

if this $\in \mathsf{LV}(\Sigma_{pos})$ **then** $[Algorithm1 - line7]$

> **this** $\in \mathsf{LV}(\exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{f1} : 1, \mathrm{f2} : O_{\mathrm{f2}}, \mathrm{f3} : 5] * f2 \mapsto [\mathbf{this} : obj, x1 : 1, \ldots])$

if window $\notin$ **Reach**(Δ_{pos}) **then return** SAFE $[Algorithm1 - line8]$

> window $\notin \{\mathbf{this}\} \cup \{obj, f2\} \cup \{f1, f2, f3, \mathbf{this}, x1\}$

Figure 5.4: Algorithm Application for Case Study A (Phase 3)

Figure 5.4 shows that the procedure Execu_SAFE_THIS executes the codes on $line2$, $line5$, $line7$, and $line8$ of Safe_This algorithm. It starts from the line2 ([*Algorithm1-line2*]). As the Δ_{pos} is not false, the algorithm reaches the line5 ([*Algorithm1-line5*]) $\Delta pos := \Pi_{pos} \parallel \Sigma_{pos} \parallel \mathbf{I}$. Since Δ_{pos} contains the variable **this** in its $heap\ formula$ Σ_{pos}, the execution continues to the line7 ([*Algorithm1-line7*]). The safety check leads the execution to the line8 ([*Algorithm1-line8*]) and produces the results that the given program is safe as the value of **this** in the final assertion is not $window$.

Phase 4: Result

Therefore, after the execution of these three phases, the $JS^t{}_{sl}$ framework reveals that the given program is safe because the value of **this** is neither innocently nor maliciously modified to be $window$.

5.2.2 Case Study B

The case study B employs the JavaScript example (see Figure 5.5) from Gardner et al. (GMS12). This example reveals the complexity of language in terms of prototype inheritance and scope chain. There are global variables x, y, z, f, v. The variable x, y, z are initialised with value $null$. A function f declares a global variable v with integer value 4, a local variable v. The correct output value of the variable x, y, z is $undefined$, 4, and 5 respectively. .

Phase 1: Program translation

The JavaScript code in Figure 5.5 transforms into a $JS^t{}_{sl}$ program in a semantic preserving way as shown in Figure 5.6.

```
1  <script type="text/javascript">
2  x = null;
3  y = null;
4  z = null;
5  f = function(w) {
6          x = v;
7          v = 4;
8          var v;
9          y = v;
10 };
11 v = 5;
12 f(null);
13 z = v;
14 </script>
```

Figure 5.5: JavaScript for Case Study B

```
1  x = null;
2  y = null;
3  z = null;
4  f = func (w) {
5          x = v;
6          var v = 4;
7          y = v
8  };
9  v = 5;
10 g= f(null);
11 z = v
```

Figure 5.6: ${JS^t}_{sl}$ for Case Study B (Phase 1)

Phase 2: Program Analysis

According to the specification language ${Spec^t}_{sl}$ defined in Section 4.4.3, the given specifications that are requested to be proved are given as:

$$\{true \parallel emp \parallel \mathbf{I}\}$$
$$C$$
$$\{true \land x = \mathsf{undef} \land y = 4 \land z = 5 \land v = 5 \land g = \mathsf{undef} \parallel \mathrm{f} \mapsto O_{\mathrm{f}} \parallel \mathbf{I}'\}$$

where C is a sequence of ${JS^t}_{sl}$ programs.

The definition of O_{f} is:

- O_{f} represents the function object referred to by f.

$$O_{\mathrm{f}} = [body : \{x = v;\ v = 4;\ varv;\ y = v\}, params : (w), \text{@}\mathsf{proto} : \mathsf{OProto}]$$

Figure 5.7 indicates the analysis process using the set of inference rules that are defined in section 4.4.6. The analysis begins from an initial precondition $\{true \parallel emp \parallel \mathbf{I}\}$. The inference rules that are applied are [**sl-glo-assign1**], [**sl-glob-assign2**], [**sl-fun-call-dir**],[**sl-local-assign1**], and [**sl-lookup-undef**]. A summary of output in this phase is a final postcondition assertion and is given as:

$$\{\exists g \cdot true \land x = \mathsf{undef} \land y = 4 \land z = 5 \land v = 5 \land g = \mathsf{undef}$$
$$\parallel \mathrm{f} \mapsto O_{\mathrm{f}} * f' \mapsto [\text{this: } \mathsf{loc_w}, w : null, ...] \parallel \mathbf{I}' \doteq LS(f.\text{@}\mathsf{scope})\}$$

Phase 3: Algorithm Application

$$\{true \parallel emp \parallel \mathbf{I}\}$$

$$\text{x} = null; \qquad \textbf{[sl-glob-assign1]}$$

$$\{true \wedge x = null \parallel emp \parallel \mathbf{I}\}$$

$$\text{y} = null; \qquad \textbf{[sl-glob-assign1]}$$

$$\{true \wedge x = null \wedge y = null \parallel emp \parallel \mathbf{I}\}$$

$$\text{z} = null; \qquad \textbf{[sl-glob-assign1]}$$

$$\{true \wedge x = null \wedge y = null \wedge z = null \parallel emp \parallel \mathbf{I}\}$$

$$\text{f} = func(w)\{...\}; \qquad \textbf{[sl-glob-assign2]}$$

$$\{true \wedge x = null \wedge y = null \wedge z = null \parallel \text{f} \mapsto O_{\text{f}} \parallel \mathbf{I'} \doteq LS(f.\textsf{@scope})\}$$

$$\text{v} = 5; \qquad \textbf{[sl-glob-assign1]}$$

$$\{true \wedge x = null \wedge y = null \wedge z = null \wedge v = 5 \parallel \text{f} \mapsto O_{\text{f}} \parallel \mathbf{I'} \doteq LS(f.\textsf{@scope})\}$$

$$\text{g} = f(null); \qquad \textbf{[sl-fun-call-dir]}$$

$$\{true \wedge x = null \wedge y = null \wedge z = null \wedge v = 5 \parallel \text{f} \mapsto O_{\text{f}} * f' \mapsto [\text{this: } \textsf{loc}_{\textsf{w}}, w : null, ...] \parallel \mathbf{I'} \doteq LS(f.\textsf{@scope})\}$$

$$\text{x} = v;\ v = 4;\ var\ v;\ y = v \qquad \textbf{[sl-glob-assign1]}$$

$$\{true \wedge x = \textsf{undef} \wedge y = 4 \wedge z = null \wedge v = 5 \parallel \text{f} \mapsto O_{\text{f}} * f' \mapsto [\text{this: } \textsf{loc}_{\textsf{w}}, w : null, ...] \parallel \mathbf{I'} \doteq LS(f.\textsf{@scope})\} \qquad \textbf{[sl-local-assign1]}$$

[sl-lookup-undef]

$$\{\exists g \cdot true \wedge x = \textsf{undef} \wedge y = 4 \wedge z = null \wedge v = 5 \wedge g = \textsf{undef} \parallel \text{f} \mapsto O_{\text{f}} * f' \mapsto [\text{this: } \textsf{loc}_{\textsf{w}}, w : null, ...] \parallel \mathbf{I'} \doteq LS(f.\textsf{@scope})\}$$

$$\text{z} = v; \qquad \textbf{[sl-glob-assign1]}$$

$$\{\exists g \cdot true \wedge x = \textsf{undef} \wedge y = 4 \wedge z = 5 \wedge v = 5 \wedge g = \textsf{undef} \parallel \text{f} \mapsto O_{\text{f}} * f' \mapsto [\text{this: } \textsf{loc}_{\textsf{w}}, w : null, ...] \parallel \mathbf{I'} \doteq LS(f.\textsf{@scope})\}$$

Figure 5.7: Program Analysis for Case Study B (Phase 2)

In the algorithm application phase, the algorithm SAFE_THIS$(C, \Delta_{pre}, \Delta_{pos})$ is applied to examine and determine whether the given program is safe in terms of execution of **this** variable. The inputs of algorithm are C, Δ_{pre}, and Δ_{pos}, The parameter C is the given program as shown in Figure 5.6. The parameter Δ_{pre} and Δ_{pos} are:

$$\Delta_{pre} = \{true \parallel emp \parallel \mathbf{I}\}$$

$$\begin{aligned}\Delta_{pos} = \; & \{\exists g \cdot true \wedge x = \mathsf{undef} \wedge y = 4 \wedge z = 5 \wedge v = 5 \wedge g = \mathsf{undef} \\ & \parallel \mathrm{f} \mapsto O_{\mathrm{f}} * f' \mapsto [\text{this: } \mathsf{loc_w}, w : null, ...] \parallel \mathbf{I}' \doteq LS(f.@\mathsf{scope})\}\end{aligned}$$

Figure 5.8 shows that Execu_SAFE_THIS executes the codes on $line2$, $line5$, $line7$, and $line10$ of the algorithm. The result analysed at $line7$ ([*Algorithm1-line7*]) indicates that Δ_{pos} contains the variable **this** pointing to the location $\mathsf{loc_w}$ which is the location for *window* object. Therefore, the line10 ([*Algorithm1-line10*]) shows that the object *window* can be reached.

Phase 4: Result

Therefore, after the execution of these three phases, the $JS^{t}{}_{sl}$ framework has proved that the given program is not safe because it modified the value of **this** pointing to *window* either innocently or maliciously.

procedure $Execu_SAFE_THIS(c, \Delta_{pre}, \Delta_{pos})$
if $\{\Delta_{pre}\}$ c $\{\Delta_{pos}\}$ **then** $[Algorithm1 - line2]$

if $\{true \parallel emp \parallel \mathbf{I}\}c\{\exists g \cdot true \wedge x = \mathsf{undef} \wedge y = 4 \wedge z = 5 \wedge$
$v = 5 \wedge g = \mathsf{undef} \parallel$
$\mathrm{f} \mapsto O_{\mathrm{f}} * f' \mapsto [\text{this: } \mathsf{loc_w}, w : null, ...] \parallel \mathbf{I'} \doteq LS(f.@\mathsf{scope})\}$**then**

$\Delta_{pos} := \Pi_{pos} \parallel \Sigma_{pos} \parallel \mathbf{I}$ $[Algorithm1 - line5]$

$\Delta_{pos} = \{\exists g \cdot true \wedge x = \mathsf{undef} \wedge y = 4 \wedge z = 5 \wedge v = 5 \wedge g = \mathsf{undef}$
$\parallel \mathrm{f} \mapsto O_{\mathrm{f}} * f' \mapsto [\text{this: } \mathsf{loc_w}, w : null, ...] \parallel \mathbf{I'} \doteq LS(f.@\mathsf{scope})\}$

if this $\in \mathsf{LV}(\Sigma_{pos})$ **then** $[Algorithm1 - line7]$

this $\in \mathsf{LV}(\mathrm{f} \mapsto O_{\mathrm{f}} * f' \mapsto [\text{this: } \mathsf{loc_w}, w : null, ...])$

if window $\in$ **Reach**(Δ_{pos}) **then return** NOT SAFE $[Algorithm1 - line10]$

window $\in \{\mathrm{f}, f'\} \cup \{\mathsf{this}, w\}$

Figure 5.8: Algorithm Application for Case Study B (Phase 3)

5.2.3 Case Study C

The case study C employs the example (see Figure 5.9) in the Chapter 4. This example shows the possibility of a host page compromised through the modification of **this** value.

Phase 1: Program translation

In the translation phase, Figure 5.10 shows the translated $JS^t{}_{sl}$ program. In the line 1-4, a literal object *obj* is declared to contain the fields x and $setX$. In the line 5, the statement $window.x$ returns $undefined$ because there is no such variable x declared in the global scope. In the line 5-7, the fields x and $setX$ are called through the object obj. In the line 8, it shows that a variable f and the identification $obj.setX$ are alias. In the line 6, it is a function invocation. After this invocation, it changes the variable x from a local variable to a global variable. In the line 10, the statement $obj.x$ still returns 10. In the line 11, the statement $window.x$ returns 90 is because the variable x has become a global variable.

Phase 2: Program Analysis

According to the specification language $Spec^t{}_{sl}$ defined in Section 4.4.3, the given

```
1  var obj = {
2     x : 0,
3     setX : function(n) { this.x = n; }
4  };
5  window.x;
6  obj.setX(10);
7  obj.x;
8  f = obj.setX;
9  f(90);
10 obj.x;
11 window.x;
```

Figure 5.9: JavaScript for Case Study C

```
1  obj = {
2      x : 0,
3      setX : func(n) { this.x = n }
4  };
5  n1= window.x;           //undefined
6  n2 = obj.setX(10);
7  n3 = obj.x;             //10
8  f = obj.setX;
9  n4= f(90);
10 n5 = obj.x;             //10
11 n6 = window.x           //90
```

Figure 5.10: $JS^t{}_{sl}$ for Case Study C (Phase 1)

specifications that are requested to be proved are given as:

$$\{true \parallel emp \parallel \mathbf{I}\}$$

$$C$$

$$\{n1 = \mathsf{undef} \wedge n2 = \mathsf{undef} \wedge n3 = 10 \wedge n4 = \mathsf{undef} \wedge n5 = 10 \wedge n6 = 90 \parallel$$
$$obj \mapsto [x : 10, setX : O_{\text{setX}} \parallel \mathbf{I}'\}$$

where C is a sequence of $JS^t{}_{sl}$ programs.

The definition of O_{setX} is:

- O_{setX} represents the function object referred to by setX.

$$O_{\text{setX}} = [body : \{this.x = n\}, params : (n), @\mathsf{proto} : \mathsf{OProto}]$$

Figure 5.11 indicates the analysis process using the set of inference rules that is described in section 4.4.6. The analysis begins from an initial precondition $\{true \parallel emp \parallel \mathbf{I}\}$. The inference rules that are applied contains [**sl-obj-crt-literal**], [**sl-lookup-undef**], [**sl-fun-call-obj**],[**sl-fun-call-dir**], A summary ofoutput in this phase is a final postcondition assertion and is given as:

$\{\exists \mathrm{n1}, \mathrm{n2}, \mathrm{n3}, \mathrm{n4}, \mathrm{f}, \mathrm{n5} \cdot \mathit{true} \wedge \mathrm{n1} = \mathsf{undef} \wedge \mathrm{n2} = \mathsf{undef} \wedge \mathrm{n3} = 10 \wedge \mathrm{n4} = \mathsf{undef}$

$\wedge\, \mathrm{f} = \mathsf{undef} \wedge \mathrm{n5} = 10 \wedge \mathrm{n6} = 90 \parallel \exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{x} : 10, \mathrm{setX} : O_{\mathrm{setX}}]$

$*\, \mathrm{setX} \mapsto [@\mathsf{this} : \mathsf{loc}_{\mathsf{w}}, \mathrm{n} : 90, \mathrm{x} = 90 \ldots] \parallel \mathbf{I} \doteq \ell s_f{:}LS\}.$

Phase 3: Algorithm Application

In this phase, the parameter Δ_{pre} and Δ_{pos} of algorithm SAFE_THIS$(C, \Delta_{pre}, \Delta_{pos})$ are:

$$\Delta_{pre} = \{\mathit{true} \parallel \mathit{emp} \parallel \mathbf{I}\}$$

$$\begin{aligned}\Delta_{pos} = \; & \{\exists \mathrm{n1}, \mathrm{n2}, \mathrm{n3}, \mathrm{n4}, \mathrm{f}, \mathrm{n5} \cdot \mathit{true} \wedge \mathrm{n1} = \mathsf{undef} \wedge \mathrm{n2} = \mathsf{undef} \wedge \mathrm{n3} = 10 \wedge \\ & \mathrm{n4} = \mathsf{undef} \wedge \mathrm{f} = \mathsf{undef} \wedge \mathrm{n5} = 10 \wedge \mathrm{n6} = 90 \parallel \exists \mathrm{obj} \cdot \mathrm{obj} \mapsto \\ & [\mathrm{x} : 10, \mathrm{setX} : O_{\mathrm{setX}}] * \mathrm{setX} \mapsto [@\mathsf{this} : \mathsf{loc}_{\mathsf{w}}, \mathrm{n} : 90, \mathrm{x} = 90 \ldots] \parallel \mathbf{I} \doteq \ell s_f{:}LS\}\end{aligned}$$

Figure 5.12 shows that the procedure Execu_SAFE_THIS executes the codes on $line2$, $line5$, $line7$, and $line10$ of the algorithm. It starts from the line2 ([*Algorithm1-line2*]). As the Δ_{pos} is not false, the algorithm1 reaches the line5 ([*Algorithm1-line5*]) $\Delta pos := \Pi_{pos} \parallel \Sigma_{pos} \parallel \mathbf{I}$. Since Δ_{pos} contains the variable **this** in its $heap\ formula$ Σ_{pos}, the execution continues to the line7 ([*Algorithm1-line7*]). The safety check leads the execution to the line10 ([*Algorithm1-line10*]).

Phase 4: Result

Therefore, after the execution of these three phases, the ${JS^t}_{sl}$ framework shows that the given program is not safe as the **this** variable in the assertion can be pointing to *window*.

$\{true \parallel emp \parallel \mathbf{I}\}$

obj = $\{\mathrm{x}: 0, \mathrm{setX}: func(n)\{\ldots\}\}$; **[sl-obj-crt-literal]**

$\{true \parallel \exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{x}: 0, \mathrm{setX}: O_{\mathrm{setX}}] \parallel \mathbf{I}\}$

n1 = $window.x$; **[sl-lookup-undef]**

$\{\exists \mathrm{n1} \cdot true \wedge \mathrm{n1} = \mathsf{undef} \parallel \exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{x}: 0, \mathrm{setX}: O_{\mathrm{setX}}] \parallel \mathbf{I}\}$

n2 = $obj.setX(10)$; **[sl-fun-call-obj]**

$\{\exists \mathrm{n1}, \mathrm{n2} \cdot true \wedge \mathrm{n1} = \mathsf{undef} \wedge \mathrm{n2} = \mathsf{undef} \parallel \exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{x}: 10, \mathrm{setX}: O_{\mathrm{setX}}] *$
$\mathrm{setX} \mapsto [\mathsf{@this}: \mathrm{obj}, \mathrm{n}: 10, \mathrm{x} = 10 \ldots] \parallel \mathbf{I} \doteq \ell s_{setX}{:}LS\}$

n3 = $obj.x$; **[sl-lookup-field]**

$\{\exists \mathrm{n1}, \mathrm{n2}, \mathrm{n3} \cdot true \wedge \mathrm{n1} = \mathsf{undef} \wedge \mathrm{n2} = \mathsf{undef} \wedge \mathrm{n3} = 10 \parallel$
$\exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{x}: 10, \mathrm{setX}: O_{\mathrm{setX}}] * \mathrm{setX} \mapsto [\mathsf{@this}: \mathrm{obj}, \mathrm{n}: 10, \mathrm{x} = 10 \ldots] \parallel \mathbf{I}\}$

f = $obj.setX$;
n4 = $f(90)$; **[sl-fun-call-dir]**

$\{\exists \mathrm{n1}, \mathrm{n2}, \mathrm{n3} \cdot true \wedge \mathrm{n1} = \mathsf{undef} \wedge \mathrm{n2} = \mathsf{undef} \wedge \mathrm{n3} = 10 \parallel$
$\exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{x}: 10, \mathrm{setX}: O_{\mathrm{setX}}] * \mathrm{setX} \mapsto [\mathsf{@this}: \mathrm{obj}, \mathrm{n}: 10, \mathrm{x} = 10 \ldots] \parallel \mathbf{I}\}$

> this.x = n; **[sl-mutate-field]**
>
> $\{\exists \mathrm{n1}, \mathrm{n2}, \mathrm{n3} \cdot true \wedge \mathrm{n1} = \mathsf{undef} \wedge \mathrm{n2} = \mathsf{undef} \wedge \mathrm{n3} = 10 \parallel$
> $\exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{x}: 10, \mathrm{setX}: O_{\mathrm{setX}}] * \mathrm{setX} \mapsto [\mathsf{@this}: \mathsf{loc_w}, \mathrm{n}: 90, \mathrm{x} = 90 \ldots] \parallel \mathbf{I}\}$

$\{\exists \mathrm{n1}, \mathrm{n2}, \mathrm{n3}, \mathrm{n4}, \mathrm{f} \cdot true \wedge \mathrm{n1} = \mathsf{undef} \wedge \mathrm{n2} = \mathsf{undef} \wedge \mathrm{n3} = 10 \wedge \mathrm{n4} = \mathsf{undef}$
$\wedge \mathrm{f} = \mathsf{undef} \parallel \exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{x}: 10, \mathrm{setX}: O_{\mathrm{setX}}]$
$* \mathrm{setX} \mapsto [\mathsf{@this}: \mathsf{loc_w}, \mathrm{n}: 90, \mathrm{x} = 90 \ldots] \parallel \mathbf{I} \doteq \ell s_f{:}LS\}$

n5 = $obj.x$; **[sl-lookup-field]**

$\{\exists \mathrm{n1}, \mathrm{n2}, \mathrm{n3}, \mathrm{n4}, \mathrm{f}, \mathrm{n5} \cdot true \wedge \mathrm{n1} = \mathsf{undef} \wedge \mathrm{n2} = \mathsf{undef} \wedge \mathrm{n3} = 10 \wedge \mathrm{n4} = \mathsf{undef}$
$\wedge \mathrm{f} = \mathsf{undef} \wedge \mathrm{n5} = 10 \parallel \exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{x}: 10, \mathrm{setX}: O_{\mathrm{setX}}]$
$* \mathrm{setX} \mapsto [\mathsf{@this}: \mathsf{loc_w}, \mathrm{n}: 90, \mathrm{x} = 90 \ldots] \parallel \mathbf{I} \doteq \ell s_f{:}LS\}$

n6 = $window.x$; **[sl-lookup-field]**

$\{\exists \mathrm{n1}, \mathrm{n2}, \mathrm{n3}, \mathrm{n4}, \mathrm{f}, \mathrm{n5} \cdot true \wedge \mathrm{n1} = \mathsf{undef} \wedge \mathrm{n2} = \mathsf{undef} \wedge \mathrm{n3} = 10 \wedge \mathrm{n4} = \mathsf{undef}$
$\wedge \mathrm{f} = \mathsf{undef} \wedge \mathrm{n5} = 10 \wedge \mathrm{n6} = 90 \parallel \exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{x}: 10, \mathrm{setX}: O_{\mathrm{setX}}]$
$* \mathrm{setX} \mapsto [\mathsf{@this}: \mathsf{loc_w}, \mathrm{n}: 90, \mathrm{x} = 90 \ldots] \parallel \mathbf{I} \doteq \ell s_f{:}LS\}$

Figure 5.11: Program Analysis for Case Study C (Phase 2)

procedure $Execu_SAFE_THIS(c, \Delta_{pre}, \Delta_{pos})$
if $\{\Delta_{pre}\}$ c $\{\Delta_{pos}\}$ **then** $[Algorithm1 - line2]$

$\textbf{if } \{true \parallel emp \parallel \mathbf{I}\}c\{\exists \mathrm{n1}, \mathrm{n2}, \mathrm{n3}, \mathrm{n4}, \mathrm{f}, \mathrm{n5} \cdot true \wedge \mathrm{n1} = \mathsf{undef} \wedge \mathrm{n2} = \mathsf{undef} \wedge$
$\mathrm{n3} = 10 \wedge \mathrm{n4} = \mathsf{undef} \wedge \mathrm{f} = \mathsf{undef} \wedge \mathrm{n5} = 10 \wedge \mathrm{n6} = 90 \parallel$
$\exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{x} : 10, \mathrm{setX} : O_{\mathrm{setX}}] * \mathrm{setX} \mapsto [@\mathsf{this} : \mathsf{loc_w}, \mathrm{n} : 90, \mathrm{x} = 90 \ldots]$
$\parallel \mathbf{I} \doteq \ell s_f{:}LS\}\textbf{then}$

$\Delta pos := \Pi_{pos} \parallel \Sigma_{pos} \parallel \mathbf{I}$ $[Algorithm1 - line5]$

$\Delta_{pos} = \{\exists \mathrm{n1}, \mathrm{n2}, \mathrm{n3}, \mathrm{n4}, \mathrm{f}, \mathrm{n5} \cdot true \wedge \mathrm{n1} = \mathsf{undef} \wedge \mathrm{n2} = \mathsf{undef} \wedge$
$\mathrm{n3} = 10 \wedge \mathrm{n4} = \mathsf{undef} \wedge \mathrm{f} = \mathsf{undef} \wedge \mathrm{n5} = 10 \wedge \mathrm{n6} = 90 \parallel$
$\exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{x} : 10, \mathrm{setX} : O_{\mathrm{setX}}] * \mathrm{setX} \mapsto [@\mathsf{this} : \mathsf{loc_w}, \mathrm{n} : 90, \mathrm{x} = 90 \ldots]$
$\parallel \mathbf{I} \doteq \ell s_f{:}LS\}$

if this $\in \mathsf{LV}(\Sigma_{pos})$ **then** $[Algorithm1 - line7]$

$\textbf{this} \in \mathsf{LV}(\exists \mathrm{obj} \cdot \mathrm{obj} \mapsto [\mathrm{x} : 10, \mathrm{setX} : O_{\mathrm{setX}}] *$
$\mathrm{setX} \mapsto [@\mathsf{this} : \mathsf{loc_w}, \mathrm{n} : 90, \mathrm{x} = 90 \ldots])$

if window $\in$ **Reach**(Δ_{pos}) **then return** SAFE $[Algorithm1 - line10]$

window $\in \{obj, setX\} \cup \{x, SetX\} \cup \{\textbf{this}, n, x\}$

Figure 5.12: Algorithm Application for Case Study C (Phase 3)

5.2.4 Case Study D

The case study D employs an "eval" function to perform malicious codes. The section 2.2.1 presents the dangerous of using the *eval* function. Figure 5.13 shows that a JavaScript example which contains a global variable x, *name*, a function f, and an *eval* function. The function f is composed of a conditional statement that returns either $g(10)$ or the value of **this**. The function *eval* declares a global function g. Since a function invocation $f(20)$ returns the global object *window*, the attack codes can be executed by the *setTimeout* [1] function after zero milliseconds.

```
1  <script type="text/javascript">
2  var x = 10;
3  var name = "David";
4
5  f = function (n) {
6          if ( n <= 10 ) {
7                  return g(10); }
8          else {
9                  return this; };
10 };
11
12 document.write(name + 'has the number:'+ x*x)");
13 eval("g=function (x) { return (x*x/2) }");
14 f(20).setTimeout("alert("attack code")", 0);
15 </script>
```

Figure 5.13: JavaScript for Case Study D

Phase 1: Program translation

The input is the given JavaScript program showed in Figure 5.13. The output is a translated JS^t_{sl} program (see Figure 5.14) in a semantic preserving way. Because the constructor *eval* is not valid in the JS^t_{sl}, the translation is finished after the declaration of function f.

[1] The function *setTimeout*(*code*, *milliseconds*) defines that an evaluation of code after a specified number of milliseconds.

```
1 x = 10;
2 name = "David";
3
4 f = function (n) {
5         if ( n <= 10 ) {
6                 return g(10)}
7         else {
8                 return this }
9 };
```

Figure 5.14: ${JS^t}_{sl}$ for Case Study D (Phase 1)

Phase 2: Program Analysis

The given specifications that are requested to be proved are given as:

$$\{true \parallel emp \parallel \mathbf{I}\}$$
$$C$$
$$\{x = 10 \wedge name = \text{"}David\text{"} \wedge \mathrm{res} = \mathrm{window} \parallel \mathrm{f} \mapsto O_{\mathrm{f}} * \mathrm{g} \mapsto O_{\mathrm{g}} \parallel \mathbf{I}'\}$$

where C is a sequence of ${JS^t}_{sl}$ programs.

In order to assist the proof in the later phases, the definition of O_{f} and O_{g} are:

- O_{f} represents the function object referred to by f.

$$\begin{aligned} O_{\mathrm{f}} = [&body : \{if(\mathrm{n} <= 10)\ return\ g(10);\\ &\qquad\qquad else\ return\ \mathrm{this};\ \},\\ ¶ms : (n), @\mathsf{proto} : \mathsf{OProto}] \end{aligned}$$

- O_{g} represents the function object referred to g.

$$\begin{aligned} O_{\mathrm{g}} = [&body : return\ (x * x/2),\\ ¶ms : (x), @\mathsf{proto} : \mathsf{OProto}] \end{aligned}$$

Figure 5.15 shows that the analysis process contains the application of inference rule [sl-glob-assign1], [sl-glob-assign2]. Because the framework ${JS^t}_{sl}$ does not provide

inference rule for the constructor *eval* function, the analysis is finished after the declaration of function f. A summary of output in this phase is a final postcondition assertion and is $\{true \land \mathrm{x} = 10 \land \mathrm{name} = "David" \parallel \exists \mathrm{f} \cdot f \mapsto O_{\mathrm{f}} \parallel \mathbf{I}' \doteq LS(f.\mathsf{@scope})\}$.

$\{true \parallel emp \parallel \mathbf{I}\}$

x = 10; [sl-glob-assign1]

$\{true \land \mathrm{x} = 10 \parallel emp \parallel \mathbf{I}\}$

name = "*David*"; [sl-glob-assign1]

$\{true \land \mathrm{x} = 10 \land \mathrm{name} = "David" \parallel emp \parallel \mathbf{I}\}$

$\mathrm{f} = func(n)\{O_{\mathrm{f}}\}$; [sl-glob-assign2]

$\{true \land \mathrm{x} = 10 \land \mathrm{name} = "David" \parallel \exists \mathrm{f} \cdot f \mapsto O_{\mathrm{f}} \parallel \mathbf{I}' \doteq LS(f.\mathsf{@scope})\}$

Figure 5.15: Program Analysis for Case Study D (Phase 2)

Phase 3: Algorithm Application

In this phase, the parameter Δ_{pre} and Δ_{pos} of algorithm SAFE_THIS$(C, \Delta_{pre}, \Delta_{pos})$ are shown as:

$$\Delta_{pre} = \{true \parallel emp \parallel \mathbf{I}\}$$

$$\Delta_{pos} = \{true \land \mathrm{x} = 10 \land \mathrm{name} = "David" \parallel \exists \mathrm{f} \cdot f \mapsto O_{\mathrm{f}} \parallel \mathbf{I}' \doteq LS(f.\mathsf{@scope})\}$$

Figure 5.16 shows that the procedure Execu_SAFE_THIS executes the codes on *line*2, *line*3 of the algorithm. It starts from the line2 ([*Algorithm1-line2*]). As the Δ_{pos} is not satisfied the given specification that is provided in the phase 1, Δ_{pos}

returns false ([*Algorithm1-line2*]). The algorithm1 produces the results $fail$.

Phase 4: Result

Therefore, the framework is not able to deal the given program and show whether it is safe.

procedure $Execu_SAFE_THIS(c, \Delta_{pre}, \Delta_{pos})$
if $\{\Delta_{pre}\}$ c $\{\Delta_{pos}\}$ **then** $[Algorithm1 - line2]$

if $\{true \parallel emp \parallel \mathbf{I}\}c\{true \wedge \mathrm{x} = 10 \wedge \mathrm{name} = "David" \parallel \exists \mathrm{f} \cdot f \mapsto O_{\mathrm{f}} \parallel \mathbf{I}' \doteq LS(f.@\mathsf{scope})\}$ **then**

if Δ_{pos} = false **then return** fail $[Algorithm1 - line3]$

$fail$

Figure 5.16: Algorithm Application for Case Study D (Phase 3)

5.3 Evaluation

This section presents analysis of results from the evaluation of case studies and an evaluation of ${JS^t}_{sl}$ framework.

5.3.1 Analysis of Case Studies

To measure the performance of ${JS^t}_{sl}$ framework on the case studies, Table 5.1 shows the summarised results from the evaluation. There are three significant results. First, the program logic in the ${JS^t}_{sl}$ framework is able to model the core features including objects, functions, fields, and the complex features such as alias, prototype inheritance, scope chain. Table 5.1 gives a statistics on the number of functions, fields, and alias over

the the number of objects. The count of objects and fields shows how frequent is the update of value of a field and the changing of prototype chains. The count of functions shows how frequent is the changing of scope chains.

The second significant result is the discovery of unsafe programs. The $JS^t{}_{sl}$ framework adapts to analyse the program that explicitly manipulates the **this** variable. There are three testing cases (A, B, and C) that have been detected unsafe operations since the analysis of program shows that **this** variable is pointing to the global object *window*. For those three cases, they are proved functional correctness and safety property by at least applying seven inference rules from the $JS^t{}_{sl}$ framework. Note that, a third party guest code has the same priority to be executed as the host code that it is embedded in. A third party code can be intentionally embedded in a host code to manipulate the global object maliciously. A host code may unintentionally change the value of **this** to *window* without the interference of a guest code. The $JS^t{}_{sl}$ framework is not only able to discover the direct malicious codes from a third party, but also can detect the indirect innocent codes from a host. In the row of $Safety$ from Table 5.1, it shows that in the Case B and C, the framework produces $NOT\ SAFE$ results due to the change of **this** value to *window*. In the Case A, it produces $SAFE$ as the **this** value remains being obj. The Case D is not applicable by the framework because the testing case contains *eval* constructor that is illegal in the syntax of $JS^t{}_{sl}$.

The third significant result is the ability of $JS^t{}_{sl}$ framework to be used to analyse and detect malicious websites automatically. This can be presented by the row of $No.\ of\ Rule\ Applied$ and the $Unsafe\ Discovered$ from Table 5.1. According to the reasoning by a set of inference rules from the $JS^t{}_{sl}$ framework, it can automatically detect that the Case A is safe, both Case B and C are not safe, and Case C is not applicable. However, the suggested correction for these unsafe cases are beyond the capability of framework.

	Case A	Case B	Case C	Case D
No. of LOC	18	14	11	15
No. of Objects	11	7	10	7
No. of Functions (Func./Obj.)	2 (18%)	2 (28%)	3 (30%)	2 (29%)
No. of Fields (Field/Obj.)	7 (64%)	4 (57%)	6 (60%)	3 (43%)
No. of Alias (Alias/Obj.)	1 (9%)	0 (0%)	1 (10%)	0 (0%)
No. of Prototype Chain	3	1	2	2
No. of Scope Chain	3	1	2	2
No. of Rule Applied	9	9	7	3
Algorithm Applicable	Yes	Yes	Yes	No
Functional Correctness	Yes	Yes	Yes	N/A
Safety	Yes	No	No	N/A
Unsafe Discovered	Yes	Yes	Yes	N/A

Table 5.1: Summary of Case Studies

5.3.2 Evaluation of JS^t_{sl} Framework

To evaluate the JS^t_{sl} framework, a comparison table is presented in Table 5.2 to show the difference of this framework with other frameworks from the literature that have been discussed in Chapter 2, Section 2.2.2. This section is an evaluation of the framework JS^t_{sl}.

The evaluation is measured by two parts, the features that the framework can deal with and the problems it can resolve. In Table 5.2, the first part evaluation presents which frameworks can manipulate which language features by symbols. Symbol "✔" and "✘" indicate that a framework can manipulate and cannot manipulate a feature respectively. An ambiguous result is indicated by a symbol "?". The table highlights the percentage of applicable features over the total 22 features. There are four frameworks that have more than 80% features covered, $Guh10$, $Maf09$, $Gar12$, and JS^t_{sl}. The second part evaluation presents which frameworks can solve which problems. For these four frameworks, $Guh10$ covers 82% of features of the language who does not have any solu-

tions. $Maf09$ that covers 82% of the features of the language solves the *capability leak* and $safety$ *breach* problem. Although the $Gar12$ has the highest feature coverage 90%, it only proves the $functional$ $correctness$ property. The $JS^t{}_{sl}$ framework has 82% of feature coverage, and prove both the $functional$ $correctness$ and $safety$ properties.

For all the frameworks in Table 5.2, there are three frameworks that are chosen for further comparing evaluation, ADsafe , $Dar12$, and $Gar12$. Because ADsafe is the most widely application, $Dar12$ solves the most number of problem, and $Gar12$ has the highest features coverage. However, $JS^t{}_{sl}$ framework produces more features coverage than ADsafe and $Dar12$, and it solves more problems than $Gar12$. Although $Dar12$ is designed to be a more robust language, but it has not been widely applied in popular browsers. The $JS^t{}_{sl}$ framework is constructed based on JavaScript, which fits the browsers naturally. $Gar12$ provides better features coverage than $JS^t{}_{sl}$, and it uses a program logic for verifying the $functional$ $correctness$ of JavaScript, but $JS^t{}_{sl}$ can be used to verify not only the $functional$ $correctness$ but also the $safety$ property.

		Frameworks and Solutions													
		ADSafe	And05	Yu07	Jen09	Chu09	Sax10	Guh10	Dar12	Maf09	Rei07	Gua09	Dew10	Gar12	JS^{t}_{sl}
Features															
Obj Creation	Obj Literal	✔	✘	✘	✔	✔	✘	✔	✘	✔	✘	✘	?	✔	✔
	Obj "new" Crt.	✔	✘	✘	✔	✔	✘	✔	✘	✔	✘	✘	?	✔	✔
Function	Function Declar.	✘	✔	✘	✔	✔	✔	✔	✘	✔	✔	✔	?	✔	✘
	Function Expre.	✘	✘	✔	✘	✘	✘	✘	✘	✘	✘	✘	?	✔	✔
	Method Call	✔	✔	✘	✘	✔	✘	✔	✔	✔	✔	✘	?	✔	✔
	Global Call	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	?	✔	✔
	Nested Func.	✔	✘	✘	✘	✘	✘	✔	✔	✔	✔	✔	?	✔	✔
Field	Field Crt.	✔	✔	✘	✔	✔	✔	✔	✔	✔	✔	✔	?	✔	✔
	Field Lookup	✔	✔	✘	✔	✔	✔	✔	✔	✔	✔	✔	?	✔	✔
	Field Mutation	✔	✔	✘	✔	✔	✘	✔	✔	✔	✔	✔	?	✔	✔
Variable	Global Assign.	✘	✔	✔	✘	✘	✔	✔	✔	✔	✘	✘	?	✔	✔
	Local Assign.	✘	✔	✘	✘	✘	✘	✔	✔	✔	✘	✘	?	✔	✔
	Expre. Return	✔	✘	✘	✔	✔	✔	✔	✔	✔	✔	✔	?	✔	✔
With		✘	✘	✘	✘	✘	✘	✔	✔	✘	✔	✘	?	✔	✘
Eval		✘	✘	✘	✘	✔	✘	✘	✘	✘	✔	✘	?	✔	✘
"this" Keyword		✘	✔	✘	✘	✔	✘	✔	✔	✔	✔	✔	?	✔	✔
Array		✔	✘	✘	✘	✘	✘	✘	✘	✔	✘	✘	?	✘	✘
Iteration		✔	✘	✔	✔	✔	✔	✔	✔	✔	✔	✔	?	✔	✔
Conditional		✔	✘	✔	✔	✔	✔	✔	✔	✔	✔	✔	?	✔	✔
Prototype Inherit.		✔	✘	✘	✔	✔	✔	✔	✔	✔	✔	✔	?	✔	✔
Scope Chain		✘	✘	✘	✘	✘	✔	✔	✔	✔	✘	✘	?	✔	✔
Alias		✘	✔	✘	✘	✔	✘	✘	✔	✘	✔	✔	?	✘	✔
Total No. of Features		13	10	5	11	15	10	18	16	18	15	12	?	20	18
(Total No. of Features/22) %		59%	45%	23%	50%	68%	45%	82%	73%	82%	68%	56%	?	90%	82%
Problem															
Functional Correctness		✘	✘	✔	✘	✘	✘	✘	✘	✘	✘	✘	✔	✔	✔
Memory Leak		✘	✘	✘	✘	✘	✘	✘	✔	✘	✘	✘	✘	✘	✘
Capability Leak		✘	✘	✘	✘	✔	✘	✘	✔	✔	✔	✔	✘	✘	?
Safety Breach		✘	✘	✔	✘	✘	✘	✘	✔	✔	✔	✔	✘	✘	✔
(Total No. of Problems/4) %		0%	0%	50%	0%	25%	0%	0%	75%	50%	50%	50%	25%	25%	50%

Table 5.2: JS^{t}_{sl} vs. other frameworks

5.4 Summary

This chapter has discussed the analysis and evaluation of $JS^t{}_{sl}$ framework. The analysis results are obtained from the case studies, they show that the program logic designed in the $JS^t{}_{sl}$ can manipulate major features of JavaScript language, including *object*, *function*, *field*, *alias*, *prototype chain*, and *scope chain*. The program logic can be used to verify the functional correctness and safety property of program. They also reveal that the framework can discover unsafe programs that have potential to inject malicious code.

By comparing the $JS^t{}_{sl}$ with the state of art in formal frameworks for JavaScript, it shows that the $JS^t{}_{sl}$ has rich language features coverage. The critical property functional correctness and safety can be verified by the $JS^t{}_{sl}$. The memory leak is not reflected because the $JS^t{}_{sl}$ only focus on the safety issue of program rather than a memory problem. The $JS^t{}_{sl}$ only can be used to partially solve the capability leak.

In summary, the results of this analysis and evaluation show that the $JS^t{}_{sl}$ framework is capable of verifying the functional correctness and safety for a suitable subset of JavaScript program.

Chapter 6

Conclusion

6.1 Introduction

This thesis has built a framework for verifying both functional correctness and safety of a subset of JavaScript programs. As introduced in Chapter 1, JavaScript has been widely used to develop third party applications on websites. Research has shown that a malicious third party application is capable of attacking host pages through JavaScript applications, it is necessary to build up a framework to improve the safety of the program for websites.

However, research to date has focused on analysing dynamic features of JavaScript. For example, some work aims to define a memory model and construct operational semantics to model JavaScript. Other research concentrates on proposing type systems for JavaScript that are accompanied by semantics. These systems can distinguish the implicit types of objects. However, these semantics are defined for a small subset of JavaScript. There is work on how to create systems for solving the problem of detecting client-side code injection vulnerability. These systems can effectively detect bugs and vulnerabilities, but are designed with less expressive subset language. There is also research on building up a program logic to systematically verify the functional correctness of a much more expressive subset of JavaScript. This research utilises separation logic to model program specification that is used to formally verify the functional correctness

of program. Nevertheless, no work has constructed a framework to consistently verify the safety property of a comprehensive subset of JavaScript.

This thesis fills in the gap and aims at proposing an effective framework for verifying the functional correctness and safety of a suitable subset of JavaScript and proving the soundness of it.

6.2 Contribution

The research in this thesis reviewed the dynamic features of JavaScript. As discussed in Chapter 2, all JavaScript programs have the features of loose typing, objects, functions, closures, prototype inheritance and scope chain. These features take the safety issues as the cost to enrich the interactions between websites and clients. The state-of-the-art of detecting vulnerability on the third party JavaScript applications reviewed the mainstream approaches, including a new language development, program rewriting isolation, sandbox virtual machine, statically verified containment.

The JS_{sl} framework, which was proposed in Chapter 3, defined a suitable subset of JavaScript that has pointer-based data structures. It captures the core features of of JavaScript, including prototype inheritance, function object, and automatic object amplifying on the fly. The functional correctness verification of the JS_{sl} programs was achieved by employing the operational semantics, the specification language $Spec_{sl}$ and the set of inference rules. The specification language was developed based on a variant of separation logic. The soundness of the axiomatic system that consists of the language $Spec_{sl}$ and the inference rules was proved with respect to the underlying operational semantics.

Safety issues are caused by the **this** variable, the ${JS^t}_{sl}$ framework, which was described in Chapter 4, enriched the subset JS_{sl} to include the **this** variable. It can deal with larger subset language with the update of corresponding operational semantics,

the specification language and inference rules. This framework not only can verify the functional correctness of the ${JS^t}_{sl}$ programs, but also verify the safety property. The impact of adding **this** variable to language was considered and presented in terms of the updated operational semantics, the ${Spec^t}_{sl}$ specification language, and the updated inference rules. More features that the ${Spec^t}_{sl}$ language can manage more than $Spec_{sl}$ are alias, scope chain, and **this** variable. The soundness of this framework was proved with respect to the updated underlying operational semantics. The SAFE_THIS algorithm was constructed to produce the detection result of safety for programs.

The evaluation of the ${JS^t}_{sl}$ framework in Chapter 5 firstly employed four different case studies. The results showed that the ${JS^t}_{sl}$ framework not only detect the malicious third party applications, but also can identify the innocent applications on host websites. Secondly, the evaluation compared the expressiveness of the subset of JavaScript with the state-of-the-art of the frameworks which are used to analyse the dynamic features of JavaScript or detect vulnerabilities. The results showed that the ${JS^t}_{sl}$ framework has rich language features, and can effectively verify the functional correctness and safety of programs.

6.3 Criteria for Success

A number of criteria for success of the research were specified in Section 1.4. This section discusses the achievement of these criteria.

1. **Definition of a suitable and a safe subset of JavaScript**

   The subsets of JavaScript in this thesis are called JS_{sl} and ${JS^t}_{sl}$. As described in Section 3.2, JS_{sl} was constructed based on JavaScript conventions. Section 4.4.1 showed that the ${JS^t}_{sl}$ was constructed to include more characteristics than have been identified by the research on program safety. Both subsets languages have the essential properties of the JavaScript programming language that can be used

to develop third party applications on websites.

2. **Define Operational and Axiomatic semantics of the JavaScript subset**

   The operational semantics of JS_{sl} and $JS^{t}{}_{sl}$ subset languages were defined in Section 3.4 and Section 4.4.1 respectivelyly. To capture program correctness, the specification language $Spec_{sl}$ in Section 3.5.1, based on separation logic, allows users to define predicates to specify program correctness that they would like to verify. Another specification language $Spec^{t}{}_{sl}$ was developed in Section 4.4.3 as an extension of $Spec_{sl}$, which can deal with alias, scope chain and the **this** variable of the $JS^{t}{}_{sl}$ language. It allows users to define predicates to specify safety property. Both specification languages were developed under the same semantic domain. The underlying axiomatic semantics in Section 3.5.2 and Section 4.4.6 were constructed to automatically reason about programs with logic assertions.

3. **Definition of safety verification algorithm**

   Section 4.4.5 formally defined safe property of programs. Section 4.4.4 showed that the Safe_This algorithm has the ability of identifying safe and unsafe programs. Chapter 5 chose four case studies and applied the algorithm for evaluation. The results in Section 5.2 showed that the algorithm can analyse assertions that are produced under axiomatic semantics and produce safe or unsafe results for users.

4. **Proofs of program written in the JavaScript subset**

   The soundness proof of the framework JS_{sl} and $JS^{t}{}_{sl}$ were presented in Appendix A and Appendix B respectively. Both frameworks were proved to be sound with respect to the underlying operational semantics in Section 3.4.2 and Section 4.4.2.

6.4 Future Work

The proposed frameworks in this thesis has achieved the intended goals, but there are still many potential aspects that can be extended in the future to make a good use of them as follows:

1. **Integration with DOM**

   The $JS^t{}_{sl}$ framework can be extended and applied to a large body of programs on client-side web programming, such as DOM objects. As the DOM originated as a specification to allow JavaScript scripts to be portable among Web browsers, it provides a structural representation of the document and assist users to modify its content and visual presentation by using JavaScript. It is observed that combing JavaScript with DOM objects cannot be detected by any automated static analysis. Therefore, in the future, the $JS^t{}_{sl}$ framework can be expanded to prevent vulnerabilities that are derived from the cooperation between DOM and JavaScript from attacking websites.

2. **Fully implemented with JavaScript libraries**

   With the expanded demands for JavaScript, the development of user interfaces for applications on websites is needed. JavaScript library including *Prototype*, *jQuery*, and JavaScript widget libraries such as *Dojo*, *YUI* have been constructed to provide platforms for assisting developers. Thus, the $JS^t{}_{sl}$ framework can be expanded to integrated with these libraries and improve the safety of programs written by them.

3. **Employ new features**

   Although the $JS^t{}_{sl}$ framework can deal with most of the features of JavaScript, the challenging features such as *eval* and *with* have not been considered to be adopted into the framework. However, the use of *eval* is frequent and has potential of

invalidating any results obtained by static analysis, and many frameworks ignore such feature. More challenging features can be enriched in the ${JS^t}_{sl}$ in the future. And the soundness proof results can be extended compositionally to include more sophisticated reasoning about "eval" and "with" constructors.

4. **Compatibility in ECMScript 5**

   The ${JS^t}_{sl}$ framework is developed under the ECMScript 3 conventions, but the latest ECMScript 5 provides interesting features. It would be interesting to reason about the new features and find the connection with ECMScript 3.

6.5 Summary

This chapter summarised the entire thesis with its achieved results and potential aspects of improvement. The results present the solution to develop a verification framework of heap-manipulating script programs with functional correctness and safety properties verification, respectively. The case studies and comparison evaluation have shown that the ${JS^t}_{sl}$ framework has given an improvement in detecting malicious scripts that run on websites. Evidently, the framework provides opportunity for further enhancement for future research, including the integration with DOM, implementation with JavaScript libraries, manipulation new features, compatibility in ECMScript 5. These facets depict a roadmap for future work.

Appendix A

Appdx A

From the perspective of backwards reasoning, an axiomatic rule is utilised according to the structure of c, and premises need to be verified with similar backward verifications until all the premises are axioms or known facts. In the following cases are organised according to the structure of c.

- Case (skip).

$$\frac{}{\{\Pi \parallel \Sigma\}\textbf{skip}\{\Pi \parallel \Sigma\}} \quad \textbf{[sl-skip]}$$

Since $skip, (s,h) \rightarrow (s,h)$, it is easy to see that rule $skip$ is sound in JS_{sl} framework.

-Case (glob-assign1).

$$\frac{}{\{\Pi[e/x] \parallel \Sigma\}x = e\{\Pi \parallel \Sigma\}} \quad \textbf{[sl-glob-assign1]}$$

Take any σ such that $s, h \models \Pi[e/x] \parallel \Sigma$. We have that $x = e, (s, h) \rightarrow (s[x \mapsto v], h)$. The goal is to prove that $s[x \mapsto v], h \models \Pi \parallel \Sigma$.

As expression e is in the stack having value v, s(e) = v, then we have that $s[e \mapsto v]$. Therefore, state $s[e \mapsto v], h \models \Pi[e/x] \parallel \Sigma$. If we replace variable e with variable x, then we will always have that that $s[x \mapsto v], h \models \Pi \parallel \Sigma$.

-Case (local-assign1).

$$\frac{}{\{\Pi \parallel \Sigma[e/x]\}\textbf{var}\ x = e\{\Pi \parallel \Sigma\}}\ \textbf{[sl-local-assign1]}$$

Take any σ such that $s, h \models \Pi \parallel \Sigma[e/x]$. We have that $\textbf{var}\ x = e, (s, h) \rightarrow (s, h[\ell \mapsto r[x \mapsto v]])$. The goal is to prove that $s, h[\ell \mapsto r[x \mapsto v]] \models \Pi \parallel \Sigma$.

As local assignment $\textbf{var}\ x = e$ is declared inside of a function, location ℓ in the heap contains a record that contains the value v of variable x, thus $s, h[\ell \mapsto r[x \mapsto v]] \models \Pi \parallel \Sigma[e/x]$. In the case of $x = e$, we replace e with the variable x, then $s, h[\ell \mapsto r[x \mapsto v]] \models \Pi \parallel \Sigma$

-Case (glob-assign2)

$$\frac{r = [\textbf{body} : c, \textbf{params} : (x_1, ..., x_n), @\mathsf{proto} : \mathsf{loc_{op}}]}{\{\Pi \parallel \textbf{emp}\}x = \textbf{func}[F](x_1, ..., x_n)\{c\}\{\Pi \parallel \exists x \cdot x \mapsto r\}}\ \textbf{[sl-glob-assign2]}$$

Take any σ such that $s, h \models \Pi \parallel emp$. Then we have that $x = \textbf{func}[F](x_1, ..., x_n)\{c\}$, $(s, h) \rightarrow s[x \mapsto \ell], h[\ell \mapsto r]$. The goal is to prove that $s[x \mapsto \ell], h[\ell \mapsto r] \models \Pi \parallel \exists x \cdot x \mapsto r$.

We start from an empty heap, $s, h \models emp$, thus $dom(h) = \emptyset$. After we create a global function x, the location of the function stores in the stack, the function body, parameters and internal prototype pointer store in the record r in the heap. Thus, we

have that $h = [x \mapsto r]$, and $s[x{\mapsto}\ell], h[\ell{\mapsto}r] \models x \mapsto r$. Therefore, we will always have that $s[x{\mapsto}\ell], h[\ell{\mapsto}r] \models \Pi \parallel x{\mapsto}r$.

-Case (local-assign2)

$$\frac{\Sigma \equiv \Sigma_0 * x_0 \mapsto r_0 \quad x_0 \in \mathbf{Func} \qquad r = [\mathbf{body} : c, \mathbf{params} : (x_1, ..., x_n), @\mathsf{proto} : \mathsf{loc_{op}}] \quad x \notin \mathsf{LV}(\Sigma)}{\{\Pi \parallel \Sigma\}\mathbf{var}\ x = \mathbf{func}[F](x_1, ..., x_n)\{c\}\{\Pi \parallel \Sigma * (\exists x \cdot x{\mapsto}r)\}} \ \textbf{[sl-local-assign2]}$$

Take any σ such that $s, h \models \Pi \parallel \Sigma$. Then we have that $\mathbf{var}\ x = \mathbf{func}[F](x_1, ..., x_n)\{c\}$, $(s, h) \to s, h[\ell_0{\mapsto}r[x \mapsto \ell]])$. The goal is to prove that $s, h[\ell_0{\mapsto}r[x \mapsto \ell]] \models \Pi \parallel \Sigma * (\exists x \cdot x{\mapsto}r)$.

We start from an empty heap, $s, h \models emp$, thus $dom(h) = \emptyset$. After we create a global function x, the location of the function stores in the stack, the function body, parameters and internal prototype pointer store in the record r in the heap. Thus, we have that $h = [\ell_0{\mapsto}r]$, and $s, h[\ell_0{\mapsto}r[x \mapsto \ell]] \models x \mapsto r$. Therefore, we will always have that $s, h[\ell_0{\mapsto}r[x \mapsto \ell]] \models \Pi \parallel \Sigma * (\exists x \cdot x{\mapsto}r)$.

-Case (lookup-field)

$$\frac{r = [..., f : v, ...]}{\{\Pi \parallel x'{\mapsto}r\}x = x'.f\{(\exists x \cdot \Pi) \wedge x{=}v \parallel x'{\mapsto}r\}} \ \textbf{[sl-lookup-field]}$$

Take any σ such that $s, h \models \Pi \parallel x'{\mapsto}r$. We have that $x = x'.f, (s, h) \to (s[x \mapsto v], h)$. The goal is to prove that $s[x \mapsto v], h \models (\exists x \cdot \Pi) \wedge x{=}v \parallel x'{\mapsto}r$.

As we know that object x' has field f with value v stored in the record r. In the stack, it keeps the value v or the reference v if v is a primitive value or a reference type value respectively. Thus we have that $s[x \mapsto v] \models (\exists x \cdot \Pi) \wedge x{=}v$. Therefore, we will

always have that $s[x \mapsto v], h \models \Pi \wedge x{=}v \parallel x'{\mapsto}r$.

-Case (lookup-proto)

$$\frac{x' \notin \mathsf{LV}(\Sigma) \quad f \notin \mathsf{dom}(r) \\ r(@\mathsf{proto}) = x'' \quad \{\Pi \parallel \Sigma\}x = x''.f\{\Pi' \parallel \Sigma'\}}{\{\Pi \parallel x'{\mapsto}r * \Sigma\}x = x'.f\{\Pi' \parallel x'{\mapsto}r * \Sigma'\}} \text{[sl-lookup-proto]}$$

Take any σ such that $s, h \models \Pi \parallel x'{\mapsto}r * \Sigma$. We have that $x = x'.f, (s, h) \rightarrow (s', h')$. The goal is to prove that $s', h' \models \Pi' \parallel x'{\mapsto}r * \Sigma'$.

As we know that the object x' is not a live variable in Σ', and the record of x' does not own the field f, but it would follow its prototype pointer @proto to evaluate f in its prototype object x''. In the case of $s, h_1 \models \Sigma'$, and $s, h_2 \models x' \mapsto r$, it shares the similar proof with the rule of **[sl-lookup-field]**. Therefore we will always have that $s', h' \models \Pi' \parallel x'{\mapsto}r{*}\Sigma'$.

-Case (lookup-undef)

$$\frac{\Sigma \equiv x' \mapsto \mathsf{OProto} \\ f \notin \mathsf{LV}(\Sigma)}{\{\Pi \parallel \Sigma\}x = x'.f\{\exists x \cdot \Pi \wedge x{=}\mathsf{undef} \parallel \Sigma\}} \text{[sl-lookup-undef]}$$

Take any σ such that $s, h \models \Pi \parallel \Sigma$. We have that $x = x'.f, (s, h) \mapsto (s[x \mapsto \mathsf{undef}], h)$. The goal is to prove that $(s[x \mapsto \mathsf{undef}], h) \models \exists x \cdot \Pi \wedge x{=}\mathsf{undef} \parallel \Sigma$.

Because even the object OProto does not have that the field f. The heap does not modified but we add to the stack with $[x \mapsto \mathsf{undef}]$. Thus we have that $s[x \mapsto \mathsf{undef}] \models \exists x{\cdot}\Pi{\wedge}x{=}\mathsf{undef}$. Therefore, we will always have that $s[x \mapsto \mathsf{undef}], h \models \exists x{\cdot}\Pi{\wedge}x{=}\mathsf{undef} \parallel \Sigma$.

-Case (mutate-field)

$$\frac{r = [..., f : v, ...] \ \ \textbf{or else} \ \ f \notin \mathsf{dom}(r)}{\{\Pi \parallel x{\mapsto}r\} x.f = ee \{\Pi \parallel x{\mapsto}r[f{\mapsto}ee]\}} \ \text{[sl-mutate-field]}$$

Take any σ such that $s, h \models \Pi \parallel x{\mapsto}r$. We have that $x.f = ee, (s, h) \to (s, h[\ell{\mapsto}r[f{\mapsto}v]])$. The goal is to prove that $s, h \models [\ell{\mapsto}r[f{\mapsto}v]] \models \Pi \parallel x{\mapsto}r[f{\mapsto}ee]$.

We know that there is $x \mapsto \ell$ in the stack, $\ell \mapsto r$ in the heap. Thus we have that $h[\ell \mapsto r[f \mapsto s(ee)] \models x \mapsto r[f \mapsto ee]$. The evaluation of the expression ee is the value v, thus we update the expression $s(ee)$ to v. Therefore we will always have that $s, h[\ell{\mapsto}r[f{\mapsto}v]] \models \Pi \parallel x{\mapsto}r[f{\mapsto}ee]$.

-Case (return)

$$\frac{}{\{\Pi[e/result] \parallel \Sigma\} return \ e \{\Pi \parallel \Sigma\}} \ \text{[sl-return]}$$

The proof for rule [sl-return] shares the same proof of the [sl-glob-assign1].

-Case (fun-call-obj)

$$\frac{\begin{array}{c}\Sigma \equiv \Sigma_0 * x'{\mapsto}[x_0{:}x'', ..] * x''{\mapsto}[\textbf{body} : c, \textbf{params} : (x_1..x_n), ...] \\ \Sigma_1 \equiv \Sigma * x_0 \mapsto [\textbf{this} : x', x_1 : e_1, ..., x_n : e_n, ...] \\ \{\Pi \parallel \Sigma_1\} c \{\Pi_1 \parallel \Sigma_2\}\end{array}}{\{\Pi \parallel \Sigma\} x = x'.x_0([e_1, ..., e_n]) \{(\exists x \cdot \Pi_1) \wedge x = \textbf{res} \parallel \Sigma_2\}} \ \text{[sl-fun-call-obj]}$$

Take any σ such that $s, h \models \Pi \parallel \Sigma$. We have that $x = x'.x_0([e_1, ..., e_n]), (s, h) \to (s_1, h_2)$. The goal is to prove that $s_1, h_2 \models (\exists x \cdot \Pi_1) \wedge x = \textbf{res} \parallel \Sigma_2$.

We know that $s, h \models \Sigma_0 * x'{\mapsto}[x_0{:}x'', ..] * x''{\mapsto}[\textbf{body} : c, \textbf{params} : (x_1..x_n), ...]$.

Thus we have that the function x_0 stored in a record in the heap, and this record is referenced by the object x'. In the case of $h_2 = h\ h_1$ and $s_1 = s[x \mapsto s\acute{(}\mathbf{result})]$, after the execution of function body c, then we have that $s_1 \models \exists x \cdot \Pi) \wedge x = \mathbf{res}$, and $h_2 \models \Sigma * x_0 \mapsto [\mathbf{this} : x', x_1 : e_1, ..., x_n : e_n, ...]$. Therefore, we will always have that $s_1, h_2 \models (\exists x \cdot \Pi_1) \wedge x = \mathbf{res} \parallel \Sigma_2$.

-Case (fun-call-proto)

$$\frac{x' \notin \mathsf{LV}(\Sigma) \quad x_0 \notin \mathsf{dom}(r) \quad r(@\mathsf{proto}) = x'' \qquad \{\Pi \parallel \Sigma\} x = x''.x_0([e_1, ..., e_n]) \{\Pi' \parallel \Sigma'\}}{\{\Pi \parallel x' \mapsto r * \Sigma\} x = x'.x_0([e_1, ..., e_n]) \{\Pi' \parallel \Sigma'\}} \ \text{[sl-fun-call-proto]}$$

Take any σ such that $s, h \models \Pi \parallel x' \mapsto r{*}\Sigma$. We have that $x = x'.x_0([e_1, ..., e_n]), (s, h) \rightarrow (s', h')$. The goal is to prove that $s', h' \models \Pi' \parallel \Sigma'$.

As we know that the function x_0 cannot be located in the object x', thus by following the internal property @proto, it leads to the location of the object x'' that is the prototype of x'. Then, the rest proof shares the similarity with the rule of [sl-fun-call-obj]. Therefore we will always have that $(s', h') \models \Pi' \parallel \Sigma'$.

-Case (func-undef)

$$\frac{\Sigma \equiv x' \mapsto \mathsf{OProto} \qquad x_0 \notin \mathsf{LV}(\Sigma)}{\{\Pi \parallel \Sigma\} x = x'.x_0([e_1, ..., e_n]) \{(\exists x \cdot \Pi) \wedge x{=}\mathsf{undef} \parallel \Sigma\}} \ \text{[sl-fun-undef]}$$

Take any σ such that $s, h \models \Pi \parallel \Sigma$. We have that $x = x'.x_0([e_1, ..., e_n]), (s, h) \rightarrow (s[x \mapsto \mathsf{undef}], h)$. The goal is to prove that $s[x \mapsto \mathsf{undef}], h \models (\exists x \cdot \Pi) \wedge x{=}\mathsf{undef} \parallel \Sigma$.

As we know that the function x_0 cannot even be found in the object OProto, thus the function invocation returns undef value. Then we have that $s[x \mapsto \mathsf{undef}] \models (\exists x \cdot \Pi) \wedge$

x=undef. Therefore, we will always have that $s[x \mapsto \mathsf{undef}], h \models (\exists x \cdot \Pi) \wedge x$=$\mathsf{undef} \parallel \Sigma$.

-Case (fun-call-dir)

$$\frac{\begin{array}{c}\Sigma \equiv (\Sigma_1 * x_0 \mapsto [\mathbf{body} : c, \mathbf{params} : (x_1...x_n), @\mathsf{proto{:}loc_{fp}}, ...]) \\ \Sigma_1 \equiv (\Sigma * x' \mapsto [\mathbf{this} : \mathsf{loc_w}, x_1 : e_1, ..., x_n : e_n, ...]) \\ \{\Pi \parallel \Sigma_1\} c \{\Pi_1 \parallel \Sigma_2\}\end{array}}{\{\Pi \parallel \Sigma\} x = x_0([e_1, ..., e_n]) \{(\exists x \cdot \Pi_1) \wedge x = \mathbf{res} \parallel \Sigma_2\}} \text{[sl-fun-call-dir]}$$

Take any σ such that $s, h \models \Pi \parallel \Sigma$. Then we have that $x = f(e_1, ..., e_n), (s, h) \rightarrow (s_1, h_2)$. The goal is to prove that $s_1, h_2 \models (\exists x \cdot \Pi_1) \wedge x = \mathbf{res} \parallel \Sigma_2$.

We know $s, h \models \Sigma_1 * x_0 \mapsto [\mathbf{body} : c, \mathbf{params} : (x_1..x_n), @\mathsf{proto} {\mapsto} \mathsf{loc_{fp}}]$. Thus, $s, h \models x_0 \mapsto [\mathbf{body} : c, \mathbf{params} : (x_1..x_n), @\mathsf{proto} {\mapsto} \mathsf{loc_{fp}}]$. As the function x_0 stores in the heap, after the function invocation, it executes the function body c, then the heap $h_2 \models \Sigma_2$. As we also have that $s \models \Pi_1$, and $x \mapsto s'(result)$ in s_1. Thus $x = result$. According to $\{\Pi \parallel \Sigma\} c \{\Pi_1 \parallel \Sigma_1\}$, we have that $s_1 \models (\exists x \cdot \Pi_1) \wedge x = \mathbf{res}$ and $h_2 \models \Sigma_2$. Therefore we will always have that $s_1, h_2 \models (\exists x \cdot \Pi_1) \wedge x = \mathbf{res} \parallel \Sigma_2$.

-Case (obj-crt-literal)

$$\frac{r = [f_1 : e_1, ..., f_n : e_n]}{\{\Pi \parallel \mathbf{emp}\}\ x = \{f_1 : e_1, .., f_n : e_n\}\ \{\Pi \parallel \exists x \cdot x {\mapsto} r\}} \text{[sl-obj-crt-literal]}$$

Take any σ such that $s, h \models \Pi \parallel \mathbf{emp}$. Then we have that $x = \{f_1 : ee_1, ..., f_n : ee_n\}, (s, h) \rightarrow (s[x \mapsto \ell], h[\ell \mapsto r])$. The goal is to prove that $s[x \mapsto \ell], h[\ell \mapsto r] \models \Pi \parallel \exists x \cdot x {\mapsto} r$.

As we know that the object x is stored in the location ℓ in the heap, thus we have that $s[x \mapsto \ell] \models \Pi$. The record r is used to keep the literal fields and values for object x in the heap, thus $h(\ell) = [f_1 : ee_1, ..., f_n = ee_n]$. Then we have that $h[\ell \mapsto r] \models \exists x \cdot x {\mapsto} r$.

Therefore we will always have that $(s[x \mapsto \ell], h[\ell \mapsto r]) \models \Pi \parallel \exists x \cdot x \mapsto r$.

-Case (obj-crt-proto)

$$\frac{}{\{\Pi \parallel x' \mapsto r\}\ x = \mathit{new}\, x'()\ \{\Pi \parallel x' \mapsto r * (\exists x \cdot x \mapsto [@\mathsf{proto} : x'])\}}\ \textbf{[sl-obj-crt-new]}$$

Take any σ such that $s, h \models \Pi \parallel x' \mapsto r$. We have that $x = \mathit{new}\ x', (s, h) \to (s[x \mapsto \ell], h[\ell \mapsto [@\mathsf{proto} : \ell']])$. The goal is to prove that $s[x \mapsto \ell], h[\ell \mapsto [@\mathsf{proto} : \ell'] \models \Pi \parallel x \mapsto [@\mathsf{proto} : x'] * x' \mapsto r$.

In the case of $h_1 = h$, $h_2 = (\ell \mapsto [@\mathsf{proto} : \ell'])$, we only need to prove that $(s, h_2) \models \{x \mapsto [@\mathsf{proto} : x']\}$. Because $dom(h) = s(x)$, and $h(s(x)) = [..., @\mathsf{proto} : x', ...]$, thus we will always have that $(s[x \mapsto \ell], h[\ell \mapsto [@\mathsf{proto} : \ell']) \models \Pi \parallel x \mapsto [@\mathsf{proto} : x'] * x' \mapsto r$.

-Case (obj-crt-fun-construct)

$$\frac{\begin{array}{c}\Sigma \equiv (\Sigma_0 * x' \mapsto [\textbf{body} : c, \textbf{params} : (x_1..x_n), @\mathsf{proto}{:}\mathsf{loc_{op}}, \textbf{this}{:}\mathsf{loc_w}]) \\ \Sigma_1 \equiv \Sigma * (\exists x'' \cdot x'' \mapsto [\textbf{this} : x, x_1 : e_1, ...x_n : e_n, ...]) \\ \{\Pi \parallel \Sigma\} c \{\Pi_1 \parallel \Sigma_2\}\end{array}}{\{\Pi \parallel \Sigma\} x = \mathit{new}\, x'([e_1, ..., e_n]) \{\Pi_1 \parallel \Sigma_2\}}\ \textbf{[sl-obj-crt-fun]}$$

Take any σ such that $s, h \models \Pi \parallel \Sigma$. We have that $x = \mathit{new}\ x'(e_1, .., e_n), (s, h) \to (s_1, h_2)$. The goal is to prove that $s_1, h_2 \models \Pi_1 \parallel \Sigma_2$.

When we create an object by function constructor, the proof can be similar completed in the similar way as that of rule **[sl-fun-call-obj]**. The only difference is that *this* keyword points to global objet before $x = \mathit{new}\ x'(e_1, .., e_n)$ execution and points to newly created object x after the execution.

-Case (sequential, conditional)

$$\frac{\{\Pi \parallel \Sigma\}c_1\{\Pi_1 \parallel \Sigma_1\} \quad \{\Pi_1 \parallel \Sigma_1\}c_2\{\Pi_2 \parallel \Sigma_2\}}{\{\Pi \parallel \Sigma\}c_1;\ c_2\{\Pi_2 \parallel \Sigma_2\}} \quad [\textbf{sl-sequential}]$$

$$\frac{\{\Pi\wedge b \parallel \Sigma\}c_1\{\Pi_2 \parallel \Sigma_2\} \quad \{\Pi\wedge\neg b \parallel \Sigma\}c_2\{\Pi_2 \parallel \Sigma_2\}}{\{\Pi \parallel \Sigma\}\textbf{if } (b)\{c_1\}\textbf{else } \{c_2\}\{\Pi_2 \parallel \Sigma_2\}} \quad [\textbf{sl-conditional}]$$

The proof for rule (sequential, conditional) are classical and can be simply followed the Hoare Logic's proof using sequencing axiom and condition axiom respectively. Thus they are omitted here.

-Case (iteration)

$$\frac{\{\Pi\wedge b \parallel \Sigma\}c\{\Pi \parallel \Sigma\}}{\{\Pi \parallel \Sigma\}\textbf{while } (b)\{c\}\{\Pi\wedge\neg b \parallel \Sigma\}} \quad [\textbf{sl-iteration}]$$

Take any σ such that $s, h \models \Pi \parallel \Sigma$. When $s(b)$ is false, then $s(b) = false$, we can easily have that $s, h \models \Pi\wedge\neg b \parallel \Sigma$. When $s(b)$ is true, it keeps looping until $s(b)$ becomes false. Thus we will always have that $s, h \models \Pi\wedge\neg b \parallel \Sigma$.

Appendix B

Appdx B

Compared with the semantics of $Spec_{sl}$ specification language, ${Spec^{t}}_{sl}$ updated its semantic model in terms of abstract state, alias, and scope Chain. The additional semantic model and logic predicate for scope chain was shown in Figure 4.16 and Figure 4.17, respectively.

From the perspective of backwards reasoning, an axiomatic rule is utilised according to the structure of c, and premises need to be verified with similar backward verifications until all the premises are axioms or known facts. In the following cases are organised according to the structure of c.

- Case (skip).

$$\frac{}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}\mathbf{skip}\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}}\ \textbf{[sl-skip]}$$

Since $skip, (s, h) \rightarrow (s, h)$, it is easy to see that rule $skip$ is sound in ${JS^{t}}_{sl}$ framework.

-Case (glob-assign1).

$$\frac{}{\{\Pi[e/x] \parallel \Sigma \parallel \mathbf{I} \doteq LS\} x = e \{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}} \text{ [sl-glob-assign1]}$$

Take any σ such that $s, h \models \Pi[e/x] \parallel \Sigma \parallel \mathbf{I}$, thus $s, h \models \Pi[e/x] \parallel \Sigma$ and $s, h \models LS$. We have that $x = e, (s, h) \to (s[x \mapsto v], h)$. The goal is to prove that $s[x \mapsto v], h \models \Pi \parallel \Sigma \parallel \mathbf{I}$.

As expression e is in the stack having value v, s(e) = v, then we have that $s[e \mapsto v]$. Therefore, state $s[e \mapsto v], h \models \Pi[e/x] \parallel \Sigma$. If we replace variable e with variable x, then we will have that $s[x \mapsto v], h \models \Pi \parallel \Sigma$. And we know $s, h \models LS$, therefore we will have that $s[x \mapsto v], h \models \Pi \parallel \Sigma \parallel \mathbf{I}$.

-Case (local-assign1).

$$\frac{}{\{\Pi \parallel \Sigma[e/x] \parallel \mathbf{I} \doteq LS\} \mathbf{var}\ x = e \{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}} \text{ [sl-local-assign1]}$$

Take any σ such that $s, h \models \Pi \parallel \Sigma[e/x] \parallel \mathbf{I}$, then $s, h \models \Pi \parallel \Sigma[e/x]$, and $s, h \models LS$. We have that $\mathbf{var}\ x = e, (s, h) \to (s, h[\ell \mapsto r[x \mapsto v]])$. The goal is to prove that $s, h[\ell \mapsto r[x \mapsto v]] \models \Pi \parallel \Sigma \parallel \mathbf{I}$.

As local assignment $\mathbf{var}\ x = e$ is declared inside of a function, location ℓ in the heap contains a record that contains the value v of variable x, thus $s, h[\ell \mapsto r[x \mapsto v]] \models \Pi \parallel \Sigma[e/x]$. In the case of $x = e$, we replace e with the variable x, and we know $s, h \models LS$ then we will always have that $s, h[\ell \mapsto r[x \mapsto v]] \models \Pi \parallel \Sigma \parallel \mathbf{I}$

-Case (glob-assign2)

$$\frac{r = [\mathbf{body} : c, \mathbf{params} : (x_1, ..., x_n), \mathsf{@proto} : \mathsf{loc_{op}}]}{\{\Pi \parallel \mathbf{emp} \parallel \mathbf{I} \doteq LS\} x = \mathbf{func}[F](x_1, ..., x_n)\{c\} \{\Pi \parallel \exists x \cdot x {\mapsto} r \parallel \mathbf{I}' \doteq LS(x.\mathsf{@scope})\}} \text{[sl-glob-assign2]}$$

Take any σ such that $s, h \models \Pi \parallel emp \parallel \mathbf{I}$, thus $s, h \models \Pi \parallel emp$, and $s, h \models \mathbf{I}$. We have that $x = \mathbf{func}[F](x_1, ..., x_n)\{c\}$, $(s, h) \rightarrow s[x{\mapsto}\ell], h[\ell{\mapsto}r]$. The goal is to prove that $s[x{\mapsto}\ell], h[\ell{\mapsto}r] \models \Pi \parallel \exists x \cdot x{\mapsto}r \parallel \mathbf{I}'$.

We start from an empty heap, $s, h \models emp$, thus $dom(h) = \emptyset$. After we create a global function x, the location of the function stores in the stack, the function body, parameters and internal prototype pointer store in the record r in the heap. Thus, we have that $h = [x \mapsto r]$, and $s[x{\mapsto}\ell], h[\ell{\mapsto}r] \models x \mapsto r$. Then $\mathbf{I}' \doteq \ell s_x : LS$, thus we have that $s[x \mapsto \ell], h[\ell \mapsto r] \models \alpha([LLS'], \ell s_x : LS, x, \ell s)$. Therefore, we will always have that $s[x{\mapsto}\ell], h[\ell{\mapsto}r] \models \Pi \parallel \exists x \cdot x{\mapsto}r \parallel \mathbf{I}'$.

-Case (local-assign2)

$$\frac{\begin{array}{c}\Sigma \equiv \Sigma_0 * x_0 \mapsto r_0 \quad x_0 \in \mathbf{Func} \\ r = [\mathbf{body} : c, \mathbf{params} : (x_1, ..., x_n), \mathsf{@proto} : \mathsf{loc_{op}}] \\ x \notin \mathsf{LV}(\Sigma) \quad \mathbf{I} \doteq \ell s_{x_0}{:}LS\end{array}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\} \mathbf{var}\ x = \mathbf{func}[F](x_1, ..., x_n)\{c\} \{\Pi \parallel \Sigma * (\exists x \cdot x{\mapsto}r) \parallel \mathbf{I}' \doteq LS(x.\mathsf{@scope})\}} \text{[sl-local-assign2]}$$

Take any σ such that $s, h \models \Pi \parallel emp \parallel \mathbf{I}$, thus $s, h \models \Pi \parallel emp$, and $s, h \models \mathbf{I}$. We have that $\mathbf{var}\ x = \mathbf{func}[F](x_1, ..., x_n)\{c\}$, $(s, h) \rightarrow s, h[\ell_0{\mapsto}r[x{\mapsto}\ell]]$. The goal is to prove that $s, h[\ell_0{\mapsto}r[x{\mapsto}\ell]] \models \Pi \parallel \Sigma * (\exists x \cdot x{\mapsto}r) \parallel \mathbf{I}' \doteq LS(x.\mathsf{@scope})$.

We start from an empty heap, $s, h \models emp$, thus $dom(h) = \emptyset$. As we know that in the heap we have the location ℓ_0 that is the location of a existing function variable x_0 pointing to its record r. After we create a local function x inside of the function x_0,

it creates data structure in the record r, which is the function variable x pointing to the location ℓ. The location ℓ has the function body, parameters and internal prototype pointer of the variable x. Thus, we have that $s, h[\ell_0 {\mapsto} r[x {\mapsto} \ell]] \models x \mapsto r$.

Then $\mathbf{I}' \doteq \ell s_x : LS$, thus we have that $s, h[\ell_0 {\mapsto} r[x {\mapsto} \ell]] \models \alpha([LLS'], \ell s_x : LS, x, \ell s)$. Therefore, we will always have that $s, h[\ell_0 {\mapsto} r[x {\mapsto} \ell]] \models \Pi \parallel \Sigma * (\exists x \cdot x {\mapsto} r) \parallel \mathbf{I}' \doteq LS(x.\textsf{@scope})$.

-Case (lookup-field)

$$\frac{r = [..., f : v, ...]}{\{\Pi \parallel x' {\mapsto} r \parallel \mathbf{I} \doteq LS\} x = x'.f \{(\exists x \cdot \Pi) \wedge x{=}v \parallel x' {\mapsto} r \parallel \mathbf{I} \doteq LS\}} \textbf{[sl-lookup-field]}$$

Take any σ such that $s, h \models \Pi \parallel x' {\mapsto} r \parallel \mathbf{I}$, thus $s, h \models \Pi \parallel x' \mapsto r$ and $s, h \models \mathbf{I}$. We have that $x = x'.f, (s, h) \to (s[x \mapsto v], h)$. The goal is to prove that $s[x \mapsto v], h \models (\exists x \cdot \Pi) \wedge x{=}v \parallel x' {\mapsto} r \parallel \mathbf{I}$.

As we know that object x' has field f with value v stored in the record r. In the stack, it keeps the value v or the reference v if v is a primitive value or a reference type value respectively. Thus we have that $s[x \mapsto v] \models (\exists x \cdot \Pi) \wedge x{=}v$. Then we will have that $s[x \mapsto v], h \models \{\Pi \wedge x{=}v \parallel x' {\mapsto} r\}$. And we know $s, h \models \mathbf{I}$, therefore we will always have that $s[x \mapsto v], h \models (\exists x \cdot \Pi) \wedge x{=}v \parallel x' {\mapsto} r \parallel \mathbf{I}$.

-Case (lookup-proto)

$$\frac{\begin{array}{c} x' \notin \textsf{LV}(\Sigma) \quad f \notin \textsf{dom}(r) \\ r(\textsf{@proto}) = x'' \quad \{\Pi \parallel \Sigma\} x = x''.f \{\Pi' \parallel \Sigma'\} \end{array}}{\{\Pi \parallel x' {\mapsto} r * \Sigma \parallel \mathbf{I} \doteq LS\} x = x'.f \{\Pi' \parallel x' {\mapsto} r * \Sigma' \parallel \mathbf{I} \doteq LS\}} \textbf{[sl-lookup-proto]}$$

Take any σ such that $s,h \models \Pi \parallel x' \mapsto r * \Sigma \parallel \mathbf{I}$, then $s,h \models \Pi \parallel x' \mapsto r * \Sigma$, and $s,h \models \mathbf{I}$. We have that $x = x'.f, (s,h) \to (s',h')$. The goal is to prove that $s',h' \models \Pi' \parallel x' \mapsto r * \Sigma' \parallel \mathbf{I}$.

As we know that the object x' is not a live variable in Σ', and the record of x' does not own the field f, but it would follow its prototype pointer @proto to evaluate f in its prototype object x''. In the case of $s,h_1 \models \{\Sigma'\}$, and $s,h_2 \models \{x' \mapsto r\}$, it shares the similar proof with the rule of [sl-lookup-field]. And we always have $s,h \models \mathbf{I}$. Therefore we will always have that $s',h' \models \Pi' \parallel x' \mapsto r * \Sigma' \parallel \mathbf{I}$.

-Case (lookup-undef)

$$\frac{\begin{array}{c}\Sigma \equiv x' \mapsto \mathsf{OProto} \\ f \notin \mathsf{LV}(\Sigma)\end{array}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\} x = x'.f \{\exists x \cdot \Pi \wedge x{=}\mathsf{undef} \parallel \Sigma \parallel \mathbf{I} \doteq LS\}} \text{[sl-lookup-undef]}$$

Take any σ such that $(s,h) \models \Pi \parallel \Sigma \parallel \mathbf{I}$, thus $s,h \models \Pi \parallel \Sigma$, and $s,h \models \mathbf{I}$. We have that $x = x'.f, (s,h) \mapsto (s[x \mapsto \mathsf{undef}], h)$. The goal is to prove that $(s[x \mapsto \mathsf{undef}], h) \models \exists x \cdot \Pi \wedge x{=}\mathsf{undef} \parallel \Sigma \parallel \mathbf{I}$.

Because even the object OProto does not have that the field f. The heap does not modified but we add to the stack with $[x \mapsto \mathsf{undef}]$. Thus we have that $s[x \mapsto \mathsf{undef}] \models \exists x \cdot \Pi \wedge x{=}\mathsf{undef}$. And we always have $s,h \models \mathbf{I}$. Therefore, we will always have that $s[x \mapsto \mathsf{undef}], h \models \exists x \cdot \Pi \wedge x{=}\mathsf{undef} \parallel \Sigma \parallel \mathbf{I}$.

-Case (mutate-field)

$$\frac{r = [..., f : v, ...] \ \textbf{or else} \ f \notin \mathsf{dom}(r)}{\{\Pi \parallel x \mapsto r \parallel \mathbf{I} \doteq LS\} x.f = ee \{\Pi \parallel x \mapsto r[f \mapsto ee] \parallel \mathbf{I} \doteq LS\}} \text{[sl-mutate-field]}$$

Take any σ such that $s, h \models \Pi \parallel x \mapsto r \parallel \mathbf{I}$, thus $s, h \models \Pi \parallel x \mapsto r$, and $s, h \models \mathbf{I}$. We have that $e.f = ee, (s, h) \rightarrow (s, h[\ell \mapsto [f \mapsto v]])$. The goal is to prove that $s, h \models [\ell \mapsto r[f \mapsto v]] \models \Pi \parallel x \mapsto r[f \mapsto ee] \parallel \mathbf{I}$.

We know that there is $x \mapsto \ell$ in the stack, $\ell \mapsto r$ in the heap. Thus we have that $h[\ell \mapsto r[f \mapsto s(ee)] \models x \mapsto r[f \mapsto ee]$. The evaluation of the expression ee is the value v, thus we update the expression $s(ee)$ to v. And we always have $s, h \models \mathbf{I}$. Therefore we will always have that $s, h[\ell \mapsto r[f \mapsto v]] \models \Pi \parallel x \mapsto r[f \mapsto ee] \parallel \mathbf{I}$.

-Case (fun-call-obj)

$$\frac{\begin{array}{c}\Sigma \equiv \Sigma_0 * x' \mapsto [x_0{:}x'', ..] * x'' \mapsto [\mathbf{body} : c, \mathbf{params} : (x_1..x_n), ...] \\ \Sigma_1 \equiv \Sigma * x_0 \mapsto [\text{this: } x', x_1 : e_1, ..., x_n : e_n, ...] \\ \{\Pi \parallel \Sigma_1\} c \{\Pi_1 \parallel \Sigma_2\} \quad \mathbf{I} \doteq \ell s_{x_0}{:}LS\end{array}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\} x = x'.x_0([e_1, ..., e_n]) \{(\exists x \cdot \Pi_1) \wedge x = \mathbf{res} \parallel \Sigma_2 \parallel \mathbf{I}' \doteq LS\}} \ \textbf{[sl-fun-call-obj]}$$

Take any σ such that $s, h \models \Pi \parallel \Sigma \parallel \mathbf{I}$, thus $s, h \models \Pi \parallel \Sigma$, and $s, h \models \mathbf{I}$. We have that $x = x'.x_0([e_1, ..., e_n]), (s, h) \rightarrow (s_1, h_2)$. The goal is to prove that $s_1, h_2 \models (\exists x \cdot \Pi_1) \wedge x = \mathbf{res} \parallel \Sigma_2 \parallel \mathbf{I}$.

We know that $s, h \models \Sigma_0 * x' \mapsto [x_0{:}x'', ..] * x'' \mapsto [\mathbf{body} : c, \mathbf{params} : (x_1..x_n), ...]$. Thus we have that the function x_0 stored in a record in the heap, and this record is referenced by the object x'. In the case of $h_2 = h\ h_1$ and $s_1 = s[x \mapsto \acute{s}(\mathbf{result})]$, after the execution of function body c, then we have that $s_1 \models \exists x \cdot \Pi) \wedge x = \mathbf{res}$, and $h_2 \models \Sigma * x_0 \mapsto [\mathbf{this} : x', x_1 : e_1, ..., x_n : e_n, ...]$. And we know the location $\ell s_{x_0} : LS$ and always have $s, h \models \mathbf{I}$. Therefore, we will always have that $s_1, h_2 \models (\exists x \cdot \Pi_1) \wedge x = \mathbf{res} \parallel \Sigma_2 \parallel \mathbf{I}$.

-Case (fun-call-proto)

$$\frac{x' \notin \mathsf{LV}(\Sigma) \quad x_0 \notin \mathsf{dom}(r) \quad r(@\mathsf{proto}) = x'' \quad \mathbf{I} \doteq \ell s_{x_0}{:}LS \qquad \{\Pi \parallel \Sigma\} x = x''.x_0([e_1, ..., e_n])\{\Pi' \parallel \Sigma'\}}{\{\Pi \parallel x' \mapsto r * \Sigma \parallel \mathbf{I} \doteq LS\} x = x'.x_0([e_1, ..., e_n])\{\Pi' \parallel \Sigma' \parallel \mathbf{I} \doteq LS\}} \text{[sl-fun-call-proto]}$$

Take any σ such that $s, h \models \Pi \parallel x' \mapsto r * \Sigma \parallel \mathbf{I}$, thus $s, h \models \Pi \parallel x' \mapsto r * \Sigma$, and $s, h \models \mathbf{I}$. We have that $x = x'.x_0([e_1, ..., e_n]), (s, h) \to (s', h')$. The goal is to prove that $s', h' \models \Pi' \parallel \Sigma' \parallel \mathbf{I}$.

As we know that the function x_0 cannot be located in the object x', thus by following the internal property @proto, it leads to the location of the object x'' that is the prototype of x'. Then, the rest proof shares the similarity with the rule of [sl-fun-call-obj]. And we know $\ell s_{x_0} : LSandalwayshaves$, h $\models \mathbf{I}$. Therefore we will always have that $(s', h') \models \Pi' \parallel \Sigma' \parallel \mathbf{I}$.

-Case (func-undef)

$$\frac{\Sigma \equiv x' \mapsto \mathsf{OProto} \qquad x_1 \notin \mathsf{LV}(\Sigma) \quad \mathbf{I} \doteq \ell s_{x_0}{:}LS}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\} x = x'.x_0([e_1, ..., e_n])\{(\exists x \cdot \Pi) \wedge x{=}\mathsf{undef} \parallel \Sigma \parallel \mathbf{I} \doteq LS\}} \text{[sl-fun-undef]}$$

Take any σ such that $s, h \models \Pi \parallel \Sigma \parallel \mathbf{I}$,thus $s, h \models \Pi \parallel \Sigma$, and $s, h \models \mathbf{I}$. We have that $x = x'.x_0([e_1, ..., e_n]), (s, h) \to (s[x \mapsto \mathsf{undef}], h)$. The goal is to prove that $s[x \mapsto \mathsf{undef}], h \models \{(\exists x \cdot \Pi) \wedge x{=}\mathsf{undef} \parallel \Sigma\}$.

As we know that the function x_0 cannot even be found in the object OProto, thus the function invocation returns undef value. Then we have that $s[x \mapsto \mathsf{undef}] \models (\exists x \cdot \Pi) \wedge x{=}\mathsf{undef}$. And we always have $s, h \models \mathbf{I}$. Therefore, we will always have that $s[x \mapsto \mathsf{undef}], h \models (\exists x \cdot \Pi) \wedge x{=}\mathsf{undef} \parallel \Sigma \parallel \mathbf{I}$.

-Case (fun-call-dir)

$$\frac{\begin{array}{c}\Sigma \equiv (\Sigma_1 * x_0 \mapsto [\mathbf{body} : c, \mathbf{params} : (x_1...x_n), @\mathsf{proto}{:}\mathsf{loc_{fp}}, ...]) \\ \Sigma_1 \equiv (\Sigma * x' \mapsto [\text{this: } \mathsf{loc_w}, x_1 : e_1, ..., x_n : e_n, ...]) \\ \{\Pi \parallel \Sigma_1 \parallel \mathbf{I}\}c\{\Pi_1 \parallel \Sigma_2 \parallel \mathbf{I}_1\} \quad \mathbf{I} \doteq \ell s_{x_0}{:}LS\end{array}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}x = x_0([e_1, ..., e_n])\{(\exists x \cdot \Pi_1) \wedge x = \mathbf{res} \parallel \Sigma_2 \parallel \mathbf{I} \doteq LS\}} \text{[sl-fun-call-dir]}$$

Take any σ such that $s, h \models \Pi \parallel \Sigma \parallel \mathbf{I}$, thus $s, h \models \Pi \parallel \Sigma$, and $s, h \models \mathbf{I}$. Then we have that $x = f(e_1, ..., e_n), (s, h) \to (s_1, h_2)$. The goal is to prove that $s_1, h_2 \models \{(\exists x \cdot \Pi_1) \wedge x = \mathbf{res} \parallel \Sigma_2\}$.

We know $s, h \models \Sigma_1 * x_0 \mapsto [\mathbf{body} : c, \mathbf{params} : (x_1..x_n), @\mathsf{proto}{\mapsto}\mathsf{loc_{fp}}]$. Thus, $s, h \models x_0 \mapsto [\mathbf{body} : c, \mathbf{params} : (x_1..x_n), @\mathsf{proto}{\mapsto}\mathsf{loc_{fp}}]$. As the function x_0 stores in the heap, after the function invocation, it executes the function body c, then the heap $h_2 \models \Sigma_2$. As we also have that $s \models \Pi_1$, and $x \mapsto s'(result)$ in s_1. Thus $x = result$. According to $\{\Pi \parallel \Sigma\}c\{\Pi_1 \parallel \Sigma_1\}$, we have that $s_1 \models (\exists x \cdot \Pi_1) \wedge x = \mathbf{res}$ and $h_2 \models \Sigma_2$. And we know that $\ell s_{x_0} : LS$ and always have $s, h \models \mathbf{I}$. Therefore we will always have that $s_1, h_2 \models (\exists x \cdot \Pi_1) \wedge x = \mathbf{res} \parallel \Sigma_2 \parallel \mathbf{I}$.

-Case (obj-crt-literal)

$$\frac{r = [f_1 : e_1, ..., f_n : e_n]}{\{\Pi \parallel \mathbf{emp} \parallel \mathbf{I} \doteq LS\}\ x = \{f_1 : e_1, .., f_n : e_n\}\ \{\Pi \parallel \exists x \cdot x{\mapsto}r \parallel \mathbf{I} \doteq LS\}} \text{[sl-obj-crt-literal]}$$

Take any σ such that $s, h \models \Pi \parallel \mathbf{emp} \parallel \mathbf{I}$, thus $s, h \models \Pi \parallel \mathbf{emp}$ and $s, h \models \mathbf{I}$. We have that $x = \{f_1 : ee_1, ..., f_n : ee_n\}, (s, h) \to (s[x \mapsto \ell], h[\ell \mapsto r])$. The goal is to prove that $s[x \mapsto \ell], h[\ell \mapsto r] \models \{\Pi \parallel \exists x \cdot x{\mapsto}r\}$.

As we know that the object x is stored in the location ℓ in the heap, thus we have that $s[x \mapsto \ell] \models \Pi$. The record r is used to keep the literal fields and values

for object x in the heap, thus $h(\ell) = [f_1 : ee_1, ..., f_n = ee_n]$. Then we have that $h[\ell \mapsto r] \models \exists x \cdot x \mapsto r$. And we always have $s, h \models \mathbf{I}$. Therefore we will always have that $(s[x \mapsto \ell], h[\ell \mapsto r]) \models \Pi \parallel \exists x \cdot x \mapsto r \parallel \mathbf{I}$.

-Case (obj-crt-proto)

$$\frac{}{\{\Pi \parallel x' \mapsto r \parallel \mathbf{I} \doteq LS\}\ x = new\, x'()\ \{\Pi \parallel x' \mapsto r * (\exists x \cdot x \mapsto [\text{@proto} : x']) \parallel \mathbf{I} \doteq LS\}} \ \textbf{[sl-obj-crt-new]}$$

Take any σ such that $s, h \models \Pi \parallel x' \mapsto r \parallel \mathbf{I}$, thus $s, h \models \Pi \parallel x' \mapsto r$, and $s, h \models \mathbf{I}$. We have that $x = new\ x', (s, h) \to (s[x \mapsto \ell], h[\ell \mapsto [\text{@proto} : \ell']])$. The goal is to prove that $s[x \mapsto \ell], h[\ell \mapsto [\text{@proto} : \ell'] \models \{\Pi \parallel x \mapsto [\text{@proto} : x'] * x' \mapsto r\}$.

In the case of $h_1 = h$, $h_2 = (\ell \mapsto [\text{@proto} : \ell'])$, we only need to prove that $(s, h_2) \models \{x \mapsto [\text{@proto} : x']\}$. Because $dom(h) = s(x)$, and $h(s(x)) = [..., \text{@proto} : x', ...]$, and we always have $s, h \models \mathbf{I}, thuswewillalwayshavethat(\text{s[x} \mapsto \ell], h[\ell \mapsto [\text{@proto} : \ell']) \models \Pi \parallel x \mapsto [\text{@proto} : x'] * x' \mapsto r \parallel \mathbf{I}$.

-Case (obj-crt-fun-construct)

$$\frac{\begin{array}{c}\Sigma \equiv (\Sigma_0 * x' \mapsto [\mathbf{body} : c, \mathbf{params} : (x_1..x_n), \text{@proto:loc}_{\text{op}}, \text{this:loc}_{\text{w}}]) \\ \Sigma_1 \equiv \Sigma * (\exists x'' \cdot x'' \mapsto [\text{this: } x, x_1 : e_1, ...x_n : e_n, ...]) \\ \{\Pi \parallel \Sigma_1\} c \{\Pi_1 \parallel \Sigma_2\} \quad \mathbf{I} \doteq \ell s_{x'} {:} LS\end{array}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\} x = new\, x'([e_1, ..., e_n]) \{\Pi_1 \parallel \Sigma_2 \parallel \mathbf{I}' \doteq LS(x.\text{@scope})\}} \ \textbf{[sl-obj-crt-fun]}$$

Take any σ such that $s, h \models \Pi \parallel \Sigma \parallel \mathbf{I}$, thus $s, h \models \Pi \parallel \Sigma$, and $s, h \models \mathbf{I}$. We have that $x = new\ x'(e_1, .., e_n), (s, h) \to (s_1, h_2)$. The goal is to prove that $s_1, h_2 \models \Pi_1 \parallel \Sigma_2 \parallel \mathbf{I}$.

When we create an object by function constructor, the proof can be similar completed in the similar way as that of rule **[sl-fun-call-obj]**. The only difference is that *this*

keyword points to global objet before $x = new\ x'(e_1, .., e_n)$ execution and points to newly created object x after the execution. Because we always have $s, h \models \mathbf{I}$, and the location $\ell s_{x'} : LS$, therefore we will always have $s_1, h_2 \models \Pi_1 \parallel \Sigma_2 \parallel \mathbf{I}'$.

-Case (sequential, conditional)

$$\frac{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}c_1\{\Pi_1 \parallel \Sigma_1 \parallel \mathbf{I}_1 \doteq LS\} \quad \{\Pi_1 \parallel \Sigma_1 \parallel \mathbf{I}_1 \doteq LS\}c_2\{\Pi_2 \parallel \Sigma_2 \parallel \mathbf{I}_2 \doteq LS\}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}c_1;\ c_2\{\Pi_2 \parallel \Sigma_2 \parallel \mathbf{I}_2 \doteq LS\}} \text{[sl-sequential]}$$

$$\frac{\{\Pi \wedge b \parallel \Sigma \parallel \mathbf{I} \doteq LS\}c_1\{\Pi_2 \parallel \Sigma_2 \parallel \mathbf{I}_2 \doteq LS\} \quad \{\Pi \wedge \neg b \parallel \Sigma \parallel \mathbf{I} \doteq LS\}c_2\{\Pi_2 \parallel \Sigma_2 \parallel \mathbf{I}_2 \doteq LS\}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}\mathbf{if}\ (b)\{c_1\}\mathbf{else}\ \{c_2\}\{\Pi_2 \parallel \Sigma_2 \parallel \mathbf{I}_2 \doteq LS\}} \text{[sl-conditional]}$$

The proof for rule (sequential, conditional) are classical and can be simply followed the Hoare Logic's proof using sequencing axiom and condition axiom respectively. Thus they are omitted here.

-Case (iteration)

$$\frac{\{\Pi \wedge b \parallel \Sigma \parallel \mathbf{I} \doteq LS\}c\{\Pi \parallel \Sigma \parallel \mathbf{I}' \doteq LS\}}{\{\Pi \parallel \Sigma \parallel \mathbf{I} \doteq LS\}\mathbf{while}\ (b)\{c\}\{\Pi \wedge \neg b \parallel \Sigma \parallel \mathbf{I}' \doteq LS\}} \text{[sl-iteration]}$$

Take any σ such that $s, h \models \Pi \parallel \Sigma \parallel \mathbf{I}$. When $s(b)$ is false, then $s(b) = false$, we can easily have that $s, h \models \Pi \wedge \neg b \parallel \Sigma \mathbf{I}$. When $s(b)$ is true, it keeps looping until $s(b)$

becomes false. Thus we will always have that $s, h \models \Pi \wedge \neg b \parallel \Sigma \parallel \mathbf{I}'$.

References

[AB04] T. Amtoft and A. Banerjee. Information flow analysis in logical form. *11th International Symposium on Static Analysis*, 3148(ISBN: 3-540-22791-1):100–115, 2004. 42

[AGD05] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. *19th European Conference on Object-Oriented Programming, ECOOP*, 3586(ISBN: 3-540-27992-X):428–452, July 2005. 43

[Alc50] [online]ACL2 Version 5.0. Available from: http://www.cs.utexas.edu/users/moore/acl2/ [cited May 2012]. 35

[BBG+60] J.W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegsein, A.V. Wijngaarden, and M. Woodger. Report on the algorithmic language algol 60. In *Communications of the ACM*, volume 3, pages 299–314, May 1960. 34

[BBN07] A. Bortz, D. Boneh, and P. Nandy. Exposing private information by timing web applicaitons. *16th International Conference on World Wide Web*, (ISBN: 978-1-59593-654-7):621–628, 2007. 42

[BCLR04] T. Ball, B. Cook, V. Levin, and S.K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. *4th In-*

ternational Conference on Integrated Formal Methods, 2999(ISBN: 3-540-21377-5):1–20, 2004. 4

[BCO05] J. Berdine, C. Calacagno, and P.W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. *Symposium on Formal Methods for Components and Objects*, 4111(ISBN: 3-540-36749-7):115–137, 2005. 5, 35, 40

[Bez90] B. Bezier. *Software Testing Techniques*. Van Nostrand Rheinhold Company, New York, USA, second edition edition, 1990. 3

[BM04] M. Bidoit and P.D. Mosses. *CASL User Manual*. Springer, 2004. 3

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, (ISBN: 1-58113-414-2):203–213, 2001. 35, 36

[BR01] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. *8th International SPIN Workshop Model Checking Software*, 2057(ISBN: 3-540-42124-6):103–122, 2001. 35

[BR02] T. Ball and S.K. Rajamani. The slam project: Debugging system software via static analysis. *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (ISBN: 1-58113-450-9):1–3, 2002. 4

[BSB07] C. Braghin, N. Sharygina, and K. BaroneAdesi. A model checking-based approach for security policy verification of mobile systems. *Formal Aspects of Computing*, 4591:37–53, 2007. 36, 42

[Bur74] R.M. Burstall. Program proving as hand simulation with a little induction.

Internaitonal Federation for Information Processing Congress, pages 308–312, 1974. 36, 37

[Cajer] [online]The Caja JavaScript Compiler. Available from: `http://code.google.com/p/google-caja/` [cited 23rd May 2012]. 27, 28

[Carom] [online]J. Lindström. Available from: `http://my.opera.com/core/blog/2009/02/04/carakan` [cited 18th May 2012]. 25

[CDNQ07] W.N. Chin, C. David, H.H. Nguyue, and S. Qin. Automated verification of shape, size and bag properties. *12th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 307–320, July 2007. 6

[CDNQ08] W. Chin, C. David, H.H. Nguyen, and S. Qin. Enhancing modular oo verification with separation logic. *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, (ISBN: 978-1-59593-689-9):87–99, January 2008. 5, 6, 43

[CDNQ10] W.N. Chin, C. David, H.H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006–1036, 2010. 5, 6, 35, 43

[CDOY09] C. Calcagno, D. Distefano, P.W. O'Heearn, and H. Yang. Compositional shape analysis by means of BI-abduction. *Principles of Programming Languages*, 58:26, 2009. 43

[CE81] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temproal logic. *In Workshop of Logic of Programs*, 131, 1981. 4, 36

[CGJ+00] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *12th Computer Aided Verification International Conference*, 1855(ISBN: 3-540-67770-4):154–169, 2000. 36

[CGJL03] E.M. Clarke, O. Grumberg, S. Jha, and Y. Lu. Counterexample-guided abstraction refinement for symbolic model checking. *ACM*, 50(5):752–794, 2003. 36

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999. 36

[CH11] R.J. Colvin and I.J. Hayes. Structual operational semantics through contex-dependent behaviour. *Journal of Logic and Algebraic Programming*, 80(7):392–426, 2011. 38

[CHA+07] J. Condit, M. Harren, Z. Anderson, D. Gay, and G.C. Necula. Dependent types for low-level programming. *16th European Symposium on Programming Languages and Systems*, 4421(ISBN: 978-3-540-71314-2):520–535, 2007. 2

[CM99] C.Kern and M.R.Grenstreet. Formal verification in hardware design: A survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, April 1999. 35

[CMS+07] S. Chen, J. Meseguer, R. Sasse, H.J. Wang, and Y.M. Wang. A systematic approach to uncover security flaws in GUI logic. *IEEE Security and Privacy*, pages 71–85, 2007. 42

[Col98] J.S. Collofello. Introduction to software verification and validation. *Software Engineering Institute Curriculum Module (SEICM)*, 1998. 35

[Coqnt] [online]The Coq Proof Assistant. Available from: http://coq.inria.fr/ [cited May 2012]. 35

[Cro08] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008. 41, 47

[Crong] [online]Making JavaScript Safe for Advertising. Available from: http://www.adsafe.org. [cited May 2010]. 29, 41, 47

[Darge] [online]Dart Language. Available from: http://www.dartlang.org/ [cited 21st May 2012]. 3, 26

[DHF10] A. Dewald, T. Holz, and F.C. Freiling. Adsandbox: Sandboxing javascript to fight malicious webstes. *SAS (Static Analysis of International Symposium)*, pages ISBN:978–1–60558–639–7, March 22-26 2010. 28

[Dij72] E.W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972. 2, 3

[Dojit] [online]The Dojo JavaScript Toolkit. Available from: http://dojotoolkit.org/ [cited 21st May 2012]. 29

[DomOM] [online]Document Object Model (DOM). Available from: www.w3.org/DOM [cited 18th May 2012]. 25

[DOY06] D. Distefano, P.W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. *12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 3920:287–302, April 2006. 43

[Ecm09] *Final Draft Of Standard ECMA-262 5th Edition ECMAScript Language*, 2009. Available from: http://www.ecma-international.org. 17, 22, 25, 42

[Eme81] E.A. Emerson. *Branching Time Temporal Logic and the Design of Correct Cocurrent Programs*. PhD thesis, Division of Applied Sciences, Harvard Unviersity, 1981. 35, 36

[Fbjpt] [online]FBJS(Facebook JavaScript). Available from: `http://developers.facebook.com/docs/fbjs/` [cited 23rd May 2012]. 27, 28

[FGH+07] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P.D. Petkov. *XSS Exploits: Cross Site Scripting Attacks and Defense*. Syngress, 2007. 42

[Fla11] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 6th edition, 2011. 12, 16, 41, 47

[Flo67] R.W. Floyd. Assigning meanings to programs. *Symposium in Applied Mathematics*, 1967. 4, 34, 35, 37, 47

[Git12] [online]2012. Available from: `]http://news.softpedia.com/news/JavaScript-Still-the-Most-Popular-Language-Java-Gaining-292027.shtml` [cited 21st May 2012]. 12

[GL09] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for Javascript code. *18th USENIX Security Symposium*, (ISBN: 978-1-931971-69-0):151–168, 2009. 28

[GM82] J.A. Goguen and J. Meseguer. Security policies and security models. *IEEE Symposium on Security and Privacy*, pages 11–20, 1982. 42

[GMS12] P. Gardner, S. Maffeis, and G. Smith. Towards a program logic for Javascript. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, number ISBN: 978-1-4503-1083-3, pages 31–44, 2012. 35, 41, 43, 123

[GSS10] A. Guha, C. Saftoiu, and S. Shnamurthi. The essence of Javascript. *24th European Conference on Object-Oriented Programming (ECOOP)*, 6183(ISBN: 978-3-642-14106-5):126–150, 2010. 42

[Har02] J. Harrison. *Formal Verification in Industry.* Logic and Automated Reasoning Summer School, Australian National University, 2002. 34

[HH91] W.C. Hetzel and B. Hetzel. *The Complete Guide to Software Testing.* Wiley, New York, USA, second edition, 1991. 3

[HJG08] G.J. Holzmann, R. Joshi, and A. Groce. New challenges in model checking. *Springer Berlin*, 5000/2008, 2008. 36, 42

[HJMS03] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. *10th International Model Checking Software SPIN Workshop*, 2648(ISBN: 3-540-40117-2):235–239, 2003. 4

[HM04] C.A.R. Hoare and R. Milner. Grand challenge in computing research. *The British Computer Society*, 2004. 2

[Hoa69] C.A.R. Hoare. An axiomatic basis of computer programing. *Communications of the ACM*, 12(576-583), 1969. 4, 5, 34, 35, 37, 47

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *ACM*, 21(8):666–667, 1978. 3

[How87] William E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill, New York, USA, 1987. 35

[HR04] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, New York, USA, 2004. 36, 44

[HT09] P. Heidegger and P. Thiemann. Recency types for dynamically-typed object-based languages. *International Workshops on Foundations of Object-Oriented Languages, FOOL*, January 2009. 43

[HYH+04] Y.W. Huang, F. Yu, C. Hang, C.H. Tsai, D.T. Lee, and S.Y. Kuo. Securing web application code by static analysis and runtime protection. *13th international conference on World Wide Web*, (ISBN: 1-58113-844-X):40–52, 2004. 42

[IEE83] IEEE. Standard glossary of software engineering terminology. *ANSI/IEEEStd729*, 1983. 35

[IO01] S.S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. *28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 36(ISBN: 1-58113-336-7):14–26, 2001. Available from: `http://dx.doi.org/10.1145/373243.375719`, `doi:10.1145/373243.375719`. 5

[Isant] [online]The Isabelle Proof assistant. Available from: `http://isabelle.in.tum.de/` [cited May 2012]. 35

[Jac99] I. Jacobson. *The Unified Software Development Process: The Complete Guide to the Unifed Process form the Original Designers*. Addison-Wesley, 1999. 3

[Jacty] [online]Jacaranda: a subset of JavaScript designed to support object-capability security. Available from: `http://jacaranda.org/` [cited 21st May 2012]. 29

[JMG+02] T. Jim, G. Morrisett, D. Grossman, M. Hicksand J. Cheney, and Y. Wang. A safe dialect of C. *USENIX: The Advanced Computing Systems Association Annual Technical Conference*, 2002. 2

[JMT09] S.H. Jensen, A. Moller, and P. Thiemann. Type analysis for Javascript. In *16th International Symposium on Static Analysis*, volume 5673, pages 238–255, Los Angeles, August 2009. 43

[Jon03] S.P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. 2

[Jslnt] [online]The JavaScript Code Quality Tool: JSLint. Available from: http://www.jslint.com/ [cited 21st June 2012]. 30

[KK97] P. Kent and J. Kent. *Official Netscape JavaScript 1.2 Book: The Nonprogrammer's Guide to Creating Interactive Web Pages*. Number 1566046750. Ventana, 2nd edition edition, 1997. 25

[KKKJ] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. *International World Wide Web Conferences*, pages 247–256. 42

[Kro77] F. Kroger. Lar: A logic of algorithmic reasoning. *Acta Informatica*, 8(3):243–266, 1977. 36

[LDG+10] X. Leroy, D. Doligez, J. Garrigue, D. Rmy, and J. Vouillon. The objective Caml system, documentation and user's manual. INRIA, 2010. 2

[LHQ08] C. Luo, G. He, and S. Qin. A heap model for Java bytecode to support separation logic. *15th Asia-Pacific Software Engineering Conference(APSEC)*, December 2008. 6, 43

[Lin12] [online]LinkedIn passwords leaked by hackers, (7 June 2012). Available from: http://www.bbc.co.uk/news/technology-18338956 [cited 8th June 2012]. 2

[Liu11] S. Liu. Automatic specification-based testing: Challenges and possibilities. *5th IEEE International Symposium on Theoretical Aspects of Software Engineering*, (ISBN: 978-1-4577-1487-0):5–8, 2011. doi:http://doi.ieeecomputersociety.org/10.1109/TASE.2011.36. 3

[MBGL06] A. Moshchuk, T. Bragin, S.D. Gribble, and H.M. Levy. A crawler-based study of spyware on the web. *Network and Distributed System Security Symposium (NDSS)*, 2006. 42

[McC63] J. McCarthy. A basis for a mathematical theory of computation. In *Computer programming and formal systems*, pages 33–70, North-Holland, Amsterdam, 1963. 34

[McM92] K.L. McMillan. Symbolic model checking: an approach to the state explosion problem. Phd thesis, Carnegie Mellon University, 1992. 36, 42

[MHK04] R. Middelkoop, K. Huizing, and R. Kuiper. A separation logic proof system for a class-based language. In *Workshop on Logics for Resources, Processes and Programs (LRPP)*, 2004. 38, 43, 44

[MMT08] S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for Javascript. *6th Asian Symposium on Programming Languages and Systems*, 5356:307–325, 2008. Available from: http://www.doc.ic.ac.uk/~{}maffeis/aplas08.pdf. 42, 43

[MMT09a] S. Maffeis, J.C. Mitchell, and A. Taly. Isolating Javascript with filters, rewriting, and wrappers. In *ESORICS*, volume 5789, pages 505–522. LNCS, 2009. 27

[MMT09b] S. Maffeis, J.C. Mitchell, and A. Taly. Language-based isolation of untrusted Javascript. In *22nd IEEE Computer Security Foundations Symposium*, number ISBN: 978-0-7695-3712-2, pages 77–91, 2009. 27

[MMT10] S. Maffeis, J.C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *31st IEEE Symposium on Security and Privacy*, number ISBN: 978-0-7695-4035-1, pages 125–140, 2010. 27, 28

[MWB10] M.Finifter, J. Weinberger, and A. Barth. Preventing capability leaks in secure Javascript subsets. *17th Annual Network and Distributed System Security Symposium (NDSS)*, March 2010. 29

[NDQC07] H.H. Nguyen, C. David, S. Qin, and W.N. Chin. Automated verification of shape and size properties via separation logic. *8th international conference on Verification, model checking, and abstract interpretation*, 4349(ISBN: 978-3-540-69735-0):251–266, 2007. 5, 6, 40, 43

[Newch] [online]A. Vance, Times Web Ads Show Security Breach. Available from: http://www.nytimes.com/2009/09/15/technology/internet/15adco.html [cited 8th June 2012]. 2

[Nitit] [online]WebKit. Available from: http://trac.webkit.org/wiki [cited 18th May 2012]. 25

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999. 4

[Plo81] G.D. Plotkin. A structual approach to operational semantics. Technical report, Computer Science Department, Aarhus University, 1981. 38

[Plo04] G.D. Plotkin. A structual approach to operational semantics. *Logic and Algebraic Programming*, 17(139):60–61, 2004. 38

[Pnu77] A. Pnueli. The temporal logic of programs. *18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977. 35, 36

[PS58] A.J. Perlis and K. Samelson. Preliminary report: international algebraic language. In *Communications of the ACM*, volume 1, 1958. 34

[QS82] J. Quelle and J. Sifakis. Specification and verification of concurrent systems in CESAR. *In Symposium proceedings, LNCS 137*, pages 337–351, 1982. 4, 36

[RDWD07] C. Reis, J. Dunagan, H.J. Wang, and O. Dubrovsky. Browsershield: Vulnerability-driven filtering of dynamic HTML. In *ACM Transactions on the Web*, volume 1, 2007. 27, 28, 42

[Rey98] J. Reynolds. *Theoreis of Programming Languages.* Cambridge University Press, 1998. 38

[Rey00] J. Reynolds. Intuitionistic reasoning about shared mutable data structure, 2000. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.5999. 39

[Rey02] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science*, number ISBN: 0-7695-1483-9, pages 55–74. IEEE, July 2002. 5, 39

[Rey05] J.C. Reynolds. An overview of separation logic. In *Verified Software: Theories, Tools, Experiments*, pages 460–469. Springer, isbn: 978-3-540-69147-1 edition, 2005. Available from: http://dx.doi.org/10.1007/978-3-540-69149-5_49, doi:10.1007/978-3-540-69149-5_49. 39, 47

[Rhiva] [online]The Rhino open-source implementation of JavaScript written entirely in Java. Available from: http://www.mozilla.org/rhino/ [cited 18th May 2012]. 25

[RL10] K. Rustan and M. Leino. Dafny: An automatic program verifier for functional correctness. *16th Logic for Programming, Artificial Intelligence, and Reasoning International Conference*, 6355(ISBN: 978-3-642-17510-7):348–370, 2010. 5

[RLBV10] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of Javascript programs. *PLDI (SIGPLAN Conference on Programming Language Design and Implementation)*, June 2010. Available from: http://sss.cs.purdue.edu/projects/dynjs/pldi275-richards.pdf. 5, 47

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998. 38

[Sch00] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 2000. 38

[SK95] K. Slonneger and B.L. Kurtz. *Formal syntax and semantics of programming languages: a laboratory based approach*. Addison-Wesley, 1995. 37

[Spi89] J. M. Spivey. *The Z notation: A Reference Manual*. Prentice-Hall, 1989. 3

[TA05] T. Terauchi and A. Aiken. Secure information flow as a safety problem. *12th International Symposium on Static Analysis*, 3672(ISBN: 3-540-28584-9):352–367, 2005. 42

[Trane] [online]The TraceMonkey Mozilla's JavaScript engine. Available from: https://wiki.mozilla.org/JavaScript:TraceMonkey [cited 18th May 2012]. 25

[tTCoP] Axiomatic Approach to Totoal Correctness of Programs. *Z. Manna and A. Pnueli*. Acta Informatic, 3 edition. 37

[Tue09] T. Tuerk. *A Formalisation of Smallfoot in HOL*. Springer, 2009. 40

[V81ne] [online]V8 JavaScript Engine. Available from: http://code.google.com/p/v8/ [cited 18th May 2012]. 25

[Ven99] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, 1999. 2

[Whi08] A. White. *JavaScript Programmer's Reference*. Wiley, 2008. 12

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993. 38

[YCIS07] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser secruity. *34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (ISBN: 1-59593-575-4):237–249, January 2007. 43

[YW09] C. Yue and H. Wang. Characterizing insecure Javascript practices on the web. *18th International World Wide Web Conference*, (ISBN: 978-1-60558-487-4):961–970, April 2009. 42