Imperial College London
Department of Computing

A Program Logic for Verification of Security Properties of Secure ECMAScript Programs

Thomas Wood

Supervisors:
Dr. Gareth Smith
Prof. Philippa Gardner

Abstract

We present an Operational Semantics of the Secure ECMAScript (SES) language. We extend Separation Logic with a backpointer operator to permit reasoning about reachability in the object graph whilst maintaining local reasoning. We define inference rules in the extended logic for SES. Finally, we prove the correctness of the Membrane design pattern.

June 2013

Submitted in partial fulfilment of the requirements for the
MEng Degree in Computing of Imperial College London

Acknowledgements

My utmost thanks go to Gareth Smith, for introducing me to the crazy world of the JavaScript language, for the continual motivation, and for assistance in refining my often incoherent bursts of inspiration.

To Philippa Gardner, for an alternative perspective on the project, and for brightening the outlook when things looked bleak.

To Mark Miller, for the SES language, and for his enlightening views on the project during his visit to Imperial.

To my lecturers this year, for rebooting my fascination and drive for theoretical Computer *Science*.

And to those friends and acquaintances at home, around the department, and online late at night, who have been willing to engage in interesting discussions when in need of a break.

Contents

Chapter 1

Introduction

Historically, the JavaScript language was developed with very little intent for it to be used for much more than form validation on web pages. However, in recent years, web technologies have become a central part of day-to-day computer use, an ever growing number of applications now target the web as their platform of choice. This increase in popularity has forced the JavaScript language to mature considerably, through syntax, semantics, and standard library support.

It is extremely common that multiple different programs from different sources may be loaded and executed in a single environment. This combination of different code poses some interesting problems regarding code-correctness and security. JavaScript provides no easy means to isolate different modules of code from one another, all data stored is potentially accessible to any code loaded on the page, which must therefore be implicitly trusted.

Some attempts have been made to solve this problem by statically rewriting source code, or more recently by making clever use of a mixture of JavaScript's deprecated and modern language features. Examples of such attempts include AdSafe [1], FBJS, and Google Caja [23]. All of these have suffered from some security flaws in their histories as a result of insufficient understanding of the JavaScript language or knowledge of one of the myriad of differences that *legitimately* exist between common JavaScript implementations [20, 22, 28]. It is therefore beneficial to introduce more security features to the core language specification.

Secure ECMAScript (SES) is a proposed revision of the ECMAScript (ES) specification that introduces changes to the language to permit programs to be executed within a restricted context, safe from the possibility of unintended leak. Restricted execution permits fine-grained control over the permissions that untrusted executing code has by restricting the set of accessible references to other objects. This is a particularly powerful method as it is possible for unrestricted code to interact with code under restricted execution, it is in this situation that it is envisaged that lapses in the encapsulation could inadvertently occur due to programmer error.

Full program verification against a specification is a good thing – the thought processes of verification often can assist find bugs in the program or potentially, in the worst case, the specification itself. However, although significant progress has been made in recent years, full program verification is too expensive in time to be performed on all code. Often it is applied to systems where it is of high benefit, such as safety critical control systems.

It is also cost-effective to verify systems and libraries, particularly those in security sensitive contexts, that will underpin a wide variety of many other software applications and systems. Because of the rapid growth in demand for features in JavaScript, we believe that SES (or a variant of it) will soon have a considerable number of users, this drives the benefit for verifying its low-level libraries.

At present, we have a program logic capable of describing the functional properties of JavaScript

programs, however, it is not capable of expressing the security properties that SES aims to enforce. It is highly convenient to reason about the security properties of a program at the same time as reasoning about the functional properties of it, as very often one is intrinsically linked to the other. We thus need to extend the logic to be able to express both functional and security properties.

1.1 Project Aims

In chapter 3 we describe and specify the Syntax and Operational Semantics of SES language as a means to understand the language, highlight its differences from JavaScript, and for direct reference whilst implementing the inference rules for the Program Logic.

In chapter 4 we extend Separation Logic to support reasoning about security properties of SES programs. We define the Hoare triple for the new logic, the soundness properties between them and the Operational Semantics, and produce axiomatic rules for SES instructions.

In chapter 5 we prove some fundamental programming patterns used regularly in SES programs, namely the Caretaker and Membrane patterns.

Chapter 2

Background

2.1 Program Verification & Hoare Logic

This style of reasoning about the functionality and correctness of computer programs was first presented in his 1969 paper "An Axiomatic Basis for Computer Programming"[15], the principles presented were based on earlier work by Floyd[12] for flowchart representations of programs, and by others in other branches of mathematics; Hoare developed the style of the reasoning to fit textual representation of programs.

Hoare's reasoning depends on the basis that commands used in programming languages should be well-defined and thus the consequences of a computer program should be determinable directly from the program's text by deductive means. Program proofs are deduced through the application of inference rules to sets of axioms.

The main contribution of the paper was to introduce the notation, now known as a Hoare triple:

$$\{P\}C\{Q\}$$

P and Q are logical assertions and C is some command or program. The triple is interpreted by Hoare as "if the assertion P is true before initiation of a program C, then the assertion Q will be true on its completion." Other interpretations of triples are possible, as will be shown later.

The axioms selected for use in the reasoning alter the semantics of the system being reasoned about. Hoare exemplifies this by presenting axioms that apply to arithmetic under all sets of integers, followed by three alternative axioms that describe different semantics for overflow in sets of finite integers.

A number of axioms and inference rules are universal or common to nearly all imperative languages, Hoare presents four of these:

(Assign)
$$\{P[f/x]\}x := f\{P\}$$

(Consequence)
$$\frac{\{P'\}C\{Q'\} \quad P \Rightarrow P' \quad Q' \Rightarrow Q}{\{P\}C\{Q\}}$$

(Iteration)
$$\frac{P \quad \{B\}S\{P\}}{\{P\}\textbf{while } B \textbf{ do } S\{\neg B \wedge P\}}$$

(Composition)
$$\frac{\{P\}C_1\{Q\} \quad \{Q\}C_2\{R\}}{\{P\}C_1;C_2\{R\}}$$

The consequence and composition rules are straight-forward. The iteration rule is also reasonably simple once it is noted that the P term is the loop invariant. The assignment axiom is the most subtle of the set when not taken in the context of a proof. The axiom asserts that the post-condition will hold

if the *precondition* has the variable replaced by the assigned value. This axiom makes more sense if it is considered to be used whilst reasoning backwards. For example, to prove $\{\text{true}\}r := x; q := 0\{x = r + y \times q\}$:

$\{\text{true} \Rightarrow x = x + y \times 0\}$	Axiom
$\{x = x + y \times 0\}r := x\{x = r + y \times 0\}$	Assignment
$\{x = r + y \times 0\}q := 0\{x = r + y \times q\}$	Assignment
$\{\text{true}\}r := x\{x = r + y \times 0\}$	Consequence
$\{\text{true}\}r := x; q := 0\{x = r + y \times q\}$	Composition

It is noted that proofs in this format are long and tedious, but it is possible to derive more useful rules from the given ones which would somewhat reduce the length of these proofs. This is now the generally accepted method of presenting proofs in Hoare's system, so the above proof could also be presented as follows:

$\{\text{true}\}$
$\{x = x + y \times 0\}$
$r := x$
$\{x = r + y \times 0\}$
$q := 0$
$\{x = r + y \times q\}$

Indeed, in his 1971 paper [16], Hoare presents a proof in such a style for the non-trivial FIND algorithm.

2.2 Separation Logic

Whilst Hoare's methodology is suitable for reasoning about programs which only operate within the stack, it is not suitable for reasoning about programs that use the heap. This is because that data structures within the heap are more prone to being shared in some manner (for example, multiple pointers to the same heap location, or even arbitrary pointer arithmetic).

Reasoning about these shared data structures using conventional logics has been attempted by a number of researchers, but no solution was particularly adequate. The main issue with verification of shared data structures was that a global view of the data, and attempts to axiomatize use of the heap would result in assertions that rapidly grew in complexity, both in terms of length, but also from universal quantification, for the need to assert that the rest of the heap was not modified along with the location of interest.

Burstall considered a local approach to this reasoning in 1972 [7], but the approach featured little further research and was unable to reason about data structures such as doubly-linked lists [26]. Reynolds began further research into the use of a spatial logic to permit local reasoning about the heap and in collaboration with O'Hearn, Ishtiaq and Yang produced Separation Logic, a combination of the spatial Bunched Implication logic with Hoare's methods for reasoning about programs [17, 25, 27].

2.2.1 Semantics of Separating Operators

Separation Logic introduces the separating conjunction operator, $P * Q$, which can be thought of as splitting the heap into two disjoint portions, in which P holds in one and Q the other. To denote the contents of the heap, the $\mapsto$ operator is also introduced to relate heap locations with values, $(x \mapsto 4)$ denotes that the value 4 is stored in the heap location x. It is often the case in languages where the heap is directly mapped to memory that locations are integers, so the notation $(x \mapsto 4, y)$ is shorthand for $(x \mapsto 4) * (x + 1 \mapsto y)$.

The separating conjunction only enforces disjointedness of the storage locations of values, it is permitted to point to other heap locations from within the heap. For example $(x \mapsto 4, y) * (y \mapsto 3, x)$ is permissible and required to be able to build complex data structures.

$x \mapsto -$ is shorthand meaning that there is some value stored at heap location x.

The disjointedness enforced by the $*$ operator provides a very powerful means of reasoning about operations made to the heap. Take for example the assertion $x \mapsto 1 * P$ and the command $x := 2$, since we have x explicitly defined on the left of the $*$, we know that it cannot occur within P. We can therefore safely conclude that P is not modified and that the postcondition must be $x \mapsto 2 * P$.

A counterpart for implication also exists, the separating implication operator, $-\!\!*$ (commonly known as the *magic wand*). The assertion $((x \mapsto 7) -\!\!* P)$ has the meaning that if x is updated to contain the value 7, then P will be true. So: $(x \mapsto 7) * ((x \mapsto 7) -\!\!* P) \Rightarrow P$.

This operator can thus, in effect, be used to generate the weakest precondition for a command given a desired postcondition. This is often used for backwards reasoning, where a postcondition for a command is known, but the precondition is not.

In separation logic, an assertion is *pure* if it does not make reference to the heap. Expressions $P * Q$ where P is pure and Q is not, may have multiple derivations through splitting of the heap in different ways. To prevent this, it is convention to use $a \doteq b$ as an abbreviation for $a = b \wedge \circledcirc$, and similarly for $\dot{\in}$.

2.2.2 Program Reasoning

Just as Hoare's logic was built atop of classical logic, we can extend Hoare's logic to the spatial logic including the separating conjunction, the combined system is known as Separation Logic.

The first concept to translate is that of the Hoare triple, $\{P\}C\{Q\}$. For the remainder of this report, we will consider Separation Logic to use *fault-avoiding* semantics for the triple. Under these semantics, then the requirements for C are strengthened, such that "if C is executed in a state satisfying P, then it will *not fault*, and if it terminates it will do so in a state satisfying Q".

Separation Logic extends the standard Hoare inference rules and axioms, the rules for Consequence, Composition, and Simple Assignment are used without modification.

The obvious new axioms to introduce are those for heap access and modification, the following set are taken from [25].

Axioms for heap access

(Update) $\{x \mapsto -\}x := v\{x \mapsto v\}$	(Delete) $\{x \mapsto -\}\texttt{delete}(x)\{\texttt{emp}\}$
(Allocate) $\{\texttt{emp}\}x := new()\{\exists L.\, x = L \wedge L \mapsto \_\}$	(Lookup) $\{x \mapsto v\}y := [x]\{y = v \wedge x \mapsto v\}$

These axioms may be expressed in other ways, for example, in a form suitable for backwards reasoning, or with use of existential quantifiers.

There is one important new inference rule, the frame rule:

(Frame Rule)

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}} \ \text{modifies}(C) \,\#\, \text{free}(R)$$

This rule is the prime reason why Separation Logic is suitable for local reasoning. It allows assertions to be split down to the *footprint* of the command, just those heap locations that the command touches.

The side condition guarantees that the command only touches the part of the assertion which is kept, it guarantees that nothing in R can be modified by C.

As a result of this, we are safe to temporarily disregard other portions of the heap. This permits rules to be easily composed without requiring an overwhelming number of predicates specifying that other portions of the heap haven't changed.

2.3 JavaScript

JavaScript is a dynamically-typed, prototype-oriented language. The early development of the language was haphazard with rival implementations produced by Netscape and Microsoft for their browsers. The language's specification was later standardised as ECMA-262 and named ECMAScript (ES). The specification is unusual as it permits any implementation-specific extensions to the language. Even today, many JavaScript implementations do not conform to the ES specifications (as determined by the incomplete ES test suite), although this situation is slowly improving.

Although originally targeted to be a programming language that was easy to learn and suitable for non-programmers to use, the language's variable scoping rules, semantics of the `this` keyword and `with` syntax are unusual and differ from most peoples' intuitions. In addition, the ability to redefine, or *monkey-patch* many primitives that define the semantics of the language means that reasoning about programs can become impossible if some unknown code can execute within the same execution environment as the program under study.

2.3.1 Formalisation

There have been several efforts to formally specify various aspects of the JavaScript language, notable mentions include Maffeis, Mitchell and Taly's production of an operational semantics for a wide range of ES3 implementations [21], this work was continued by Gardner, Maffeis, Smith's [13] by producing a Program Logic for JavaScript which is capable of expressing JavaScript program states with high precision. Since SES is based upon JavaScript, sharing many of the simple statements, this program logic was used as a starting point for the formalisation of SES.

As with most formal models, they approximate and simplify actual behaviour where it does not impact the aspects of the language under study. JavaScript is a particularly large and awkward language, notable differences between the model and the full language are that the syntax is flattened to be purely expression-based and that the mechanism for automatic casting of types is avoided by requiring the primitive type is specified where needed in command pre-conditions.

Once again, Hoare-style reasoning is adapted to suit the extensions of the logic. The fault-avoiding semantics for the Hoare triple are maintained. The basic set of Separation Logic axioms and inference rules (excluding those referring directly to the C-style heap implementation) are used. One notable difference is that the Frame rule loses its side condition as a result of all variables now being on the heap – the footprint of C will be expressed in the P and Q terms, so is necessarily disjoint from the framed R portion of the rule.

2.3.2 Data Structure

A unique feature of the language is that the entire state is stored on the heap in a structure that loosely resembles the variable store in conventional programming languages. It thus follows that Separation Logic is likely to be a useful basis for the JavaScript program logic. However, a considerable number of complex predicates were required to accurately model the variable store and scoping rules.

A JavaScript object is identified by a location $l \in \mathcal{L}$, each object may have fields $x \in \mathcal{X}$ which map to values $v \in \mathcal{V}$. The heap is represented by the partial function of locations and fields to values, $H \in \mathcal{L} \times \mathcal{X} \rightharpoonup \mathcal{V}$. For example, $(x, y) \mapsto z$ means that the object in location x, has a field named y which points to the location z. Note that unlike the original Separation Logic semantics, tuples of values are not supported, since the heap locations are now unordered.

Field names are subdivided into user fields $\mathcal{X}^{\mathrm{U}}$ and internal fields $\mathcal{X}^{\mathrm{I}}$ (prefixed by an @ symbol). No internal fields are permitted to be directly accessed by executing code, they are used by the language to store contextual information on objects for purposes such as prototypical name resolution and function closures.

Field lookup on an object proceeds by checking whether the field is present on the object, if not the *prototypical* object stored on the @*proto* field is recursively checked for the field. The list of objects checked for field lookup is known as the *prototype chain* of an object. The object lookup operation is performed by the π predicate in the Operational Semantics and Program Logic, the corresponding value retrieval operation is performed by γ.

During execution a *scope chain*, an ordered list of object locations, is maintained. This structure is somewhat alike to stack frames in other languages. To perform a variable lookup, a prototypical field lookup is performed for each object in the scope chain in turn until a field is matched. This operation is performed by the σ predicate in the formalisations.

To perform a write to a variable, a similar lookup is performed, except the field is modified on the *scope object* for the prototype chain in which the variable was found – this is so that the prototypical variable is overridden, but not overwritten.

SES is an object-capability style of language, objects are considered to hold capabilities to privileged functionality or information. Because of this, it is essential that the language does not provide the ability to access arbitrary memory locations. Although locations are values that are stored in the heap, they may not be directly referred to by user programs. This means that references to objects are not forgeable. This is a key security concept in the SES language: an object must be passed a reference to another object to be able to access it. Since all functions (including standard libraries and resources) are themselves objects, the concept of access extends to one of privilege when a security-sensitive interface needs to be protected.

Chapter 3

SES Language Syntax & Operational Semantics

We now define the syntax and semantics of expressions in this model of the SES language. We present the English definition of each expression along with the Operational Semantics for the precise effect it has on the heap and program state. We also discuss how SES differs from JavaScript and the rationale for these differences.

3.1 Syntax

Syntax of SES values and expressions, `v`, `e`

`v` ::=	`n`	*Number*
|	`s`	*String*
|	`undefined`	*Undefined*
|	`null`	*Null*
|	`true` | `false`	*Booleans*
`e` ::=	`v`	*Value*
|	`var x`	*Variable declaration*
|	`x`	*Identifier*
|	$\{x_1 : e_1, \ldots, x_n : e_n\}$	*Object creation*
|	`e;e`	*Sequence*
|	$e \oplus e$	*Binary operator*
|	`if(e){e}else{e}`	*Conditional*
|	`while(e){e}`	*Looping*
|	`e = e`	*Assignment*
|	`e.x`	*Member access*
|	`e[e]`	*Computed member access*
|	`e(e)`	*Function call*
|	`this`	*this*
|	`function (e){e}`	*Function creation*
|	`function x(e){e}`	*Named function creation*
|	`new e(e)`	*Object construction*
|	`reval(e,e)`	*Restricted evaluation – similar to* `eval` *in JavaScript*
|	`freeze(e)`	*Freeze – make one object immutable*
|	`def(e)`	*Recursive freeze – make many objects immutable*

Numbers, strings, booleans and the special values of `undefined` and `null` are the only primitive values.

We model SES as an expression-based language, the full language follows JavaScript's Strict Mode parsing and variable declaration rules. For this project, we assume that all programs proved additionally pass these checks.

SES syntax differs from JS syntax only by removal and addition of several commands. The most notable removal is that of the `with` expression, this expression would add the given JS object into the scope chain for the duration of execution of the given block. This causes the scope chain to potentially have multiple prototype chains requiring traversal. Removal of this command ensures that all objects in the scope chain, other than the root, have no prototype chains. This removal considerably simplifies the reasoning required for variable lookup. The deprecation of the `with` syntax has also been started in the most recent ES specifications.

We are modelling three additional primitives present in the SES language, these are restricted evaluation, `reval(e,e)`, and commands to make objects immutable, `freeze(e)` and `def(e)`.

3.2 Operational Semantics

We now describe the semantics of each expression given in the syntax, both informally in English, and formally with the big-step operational semantics rule, which precisely describes how the program state is modified by execution of each expression.

Although daunting at first, the rules are quite simple to follow. The syntax $H, L, \mathtt{e} \longrightarrow H', r$ means that the heap H, scope chain L, and expression `e` will evaluate to the heap H' and return value r. (A return value is a value, or a reference, $l{\cdot}\mathtt{x}$, to a location and a field).

An expression will be evaluated iff the preconditions above the line hold, these may be evaluations of subexpressions, or involve the use of predicates that test or update the current state. Ordering of the evaluation of these preconditions is enforced through the labelling of the variables in each rule.

The semantics make use of several auxiliary functions defined in Appendix B and described above in Section 2.3.2. Recall that γ is used to dereferencing a JavaScript or SES reference value. As the execution of a subexpression followed by the dereferencing of its return reference into a value is a common activity, $\xrightarrow{\gamma}$ is defined as shorthand for this:

$$H, L, \mathtt{e} \xrightarrow{\gamma} H', v \triangleq \exists r.(H, L, \mathtt{e} \longrightarrow H', r \wedge \gamma(H', r) = v)$$

The semantics here were derived from a number of sources, we used the JavaScript program logic [13] as the baseline for all commands. Deviations from this are noted with the appropriate command.

3.2.1 Structural Expressions

All of these semantics are common across most imperative languages.

(Value)
$H, L, \mathtt{v} \longrightarrow H, v$

A value simply evaluates to itself, the heap is not modified. No preconditions are required.

(Variable declaration)
$H, L, \mathtt{var\ x} \longrightarrow H, \mathtt{undefined}$

The variable declaration expression is only checked on entry to a function body, they are treated as a no-op during execution.

(Sequence)

$$\frac{H, L, \mathtt{e_1} \longrightarrow H'', r' \quad H'', L, \mathtt{e_2} \longrightarrow H', r}{H, L, \mathtt{e_1; e_2} \longrightarrow H', r}$$

The sequencing operator specifies that the expression on the left should be evaluated, followed by the expression on the right. The result of the right subexpression is returned.

(Binary operator)

$$\frac{H, L, \mathtt{e_1} \xrightarrow{\gamma} H'', v_1 \quad H'', L, \mathtt{e_2} \xrightarrow{\gamma} H', v_2 \quad v_1 \oplus v_2 = v}{H, L, \mathtt{e_1} \oplus \mathtt{e_2} \longrightarrow H', v}$$

Binary operations are evaluated left-to-right, and include standard mathematical and string operations. Formalisation of semantics for all of these are beyond the scope of this project, but will correspond closely to similar works on the JavaScript family of languages.

(Conditional true)

$$\frac{H, L, \mathtt{e_1} \xrightarrow{\gamma} H'', v \quad \mathsf{True}(v) \quad H'', L, \mathtt{e_2} \longrightarrow H', r}{H, L, \mathtt{if(e_1)\{e_2\}else\{e_3\}} \longrightarrow H', r}$$

(Conditional false)

$$\frac{H, L, \mathtt{e_1} \xrightarrow{\gamma} H'', v \quad \mathsf{False}(v) \quad H'', L, \mathtt{e_3} \longrightarrow H', r}{H, L, \mathtt{if(e_1)\{e_2\}else\{e_3\}} \longrightarrow H', r}$$

The conditional expression evaluates the first sub-expression, if it evaluates to `true` when cast to a boolean then the second sub-expression is evaluated and result returned, otherwise the third sub-expression is evaluated and result returned.

(While true)

$$\frac{H, L, \mathtt{e_1} \xrightarrow{\gamma} H'', v \quad \mathsf{True}(v) \quad H'', L, e_2; \mathtt{while(e_1)\{e_2\}} \longrightarrow H', v''}{H, L, \mathtt{while(e_1)\{e_2\}} \longrightarrow H', \mathtt{undefined}}$$

(While false)

$$\frac{H, L, \mathtt{e_1} \xrightarrow{\gamma} H', v \quad \mathsf{False}(v)}{H, L, \mathtt{while(e_1)\{e_2\}} \longrightarrow H', \mathtt{undefined}}$$

Whilst the loop guard is true, the loop is syntactically unrolled once, and the resulting expression evaluated. When the guard condition evaluates to false, `undefined` is returned. This is the standard loop unrolling definition.

3.2.2 Heap-based Expressions

(Variable Resolution)

$$\frac{\sigma(H, L, x) = l' \quad l' \neq \mathtt{null}}{H, L, \mathtt{x} \longrightarrow H, l' \cdot x}$$

The σ predicate is used to search the scope chain for an object which contains the field matching the variable name. A reference to the *scope* object and the field is returned.

If the scope object cannot be found, fail to evaluate – this behaviour follows the ES5.1 Strict standard, a considerable difference from ES5.1 Non-Strict or earlier JS, which return the reference `null·x`. JS assigned write attempts to the `null` object to the *global* object – this posed an obvious isolation problem. This new behaviour enforces that all variables must be declared before use.

(Member access)

$$\frac{H, L, \mathtt{e} \xrightarrow{\gamma} H', l' \quad l' \neq \mathtt{null}}{H, L, \mathtt{e.x} \longrightarrow H', l' \cdot x}$$

(Computed member access)

$$\frac{H, L, \mathtt{e_1} \xrightarrow{\gamma} H'', l' \quad l' \neq \mathtt{null} \quad H'', L, \mathtt{e_2} \longrightarrow H', x}{H, L, \mathtt{e_1[e_2]} \longrightarrow H', l' \cdot x}$$

Returns a reference to the named field of the object referenced by the first subexpression. For the computed member access, the second subexpression is evaluated to a field name. In full JS and SES, these rules are additionally complicated with details about type casting. For simplicity, here we focus on the case where no type casting is necessary.

(Assignment)

$$\frac{\begin{array}{l} H, L, \mathtt{e_1} \longrightarrow H_1, l{\cdot}x \qquad \mathsf{RW}(H_1, l) \\ H_1, L, \mathtt{e_2} \xrightarrow{\gamma} H_2, v \\ H' = H_2[(l, x) \mapsto v] \end{array}}{H, L, \mathtt{e_1} = \mathtt{e_2} \longrightarrow H', v}$$

Assigns the value on the right hand side to the subexpression evaluating to a reference on the left. The object being assigned to must be writable (see definition of Freeze).

Assignments to undeclared variables are forbidden (enforced by the Variable rule).

(Object creation)

$$\frac{\begin{array}{l} H_0 = H \uplus \mathsf{obj}(l, l_{op}) \\ \forall i \in 1..n. \left(\begin{array}{l} H_{i-1}, L, \mathtt{e_i} \xrightarrow{\gamma} H'_i, v_i \\ H_i = H'_i[(l, \mathtt{x_i}) \mapsto v_i] \end{array} \right) \end{array}}{H, L, \{\mathtt{x_1} : \mathtt{e_1}, \ldots, \mathtt{x_n} : \mathtt{e_n}\} \longrightarrow H_n, l}$$

The Object Construction syntax produces a new object at the new location l, with fields named $\mathtt{x_1}, \ldots, \mathtt{x_n}$ which map to the values of expressions $\mathtt{e_1}, \ldots, \mathtt{e_n}$ when evaluated in order at creation. The location of the new object is returned.

3.2.3 Function-related Expressions

(Function creation)

$$\frac{H' = H \uplus \mathsf{obj}(l, l_{op}) \uplus \mathsf{fun}(l', L, \mathtt{x}, \mathtt{e}, l)}{H, L, \mathtt{function}\ (\mathtt{x})\{\mathtt{e}\} \longrightarrow H', l'}$$

Creates a function object, assigning the body, parameter declarations and the current scope to internal fields. The shape of this function object is given by the fun auxiliary function, defined in Appendix B. A new object is also created and assigned to the `prototype` field for use as the prototype of objects produced by using this function as a constructor.

(Named function creation)

$$\frac{H' = H \uplus \mathsf{obj}(l, l_{op}) \uplus \mathsf{fun}(l', l_1{:}L, \mathtt{x}, \mathtt{e}, l) \uplus l_1 \mapsto \{@proto : \mathtt{null}, y : l'\}}{H, L, \mathtt{function}\ \mathtt{y}(\mathtt{x})\{\mathtt{e}\} \longrightarrow H', l'}$$

This is the same as standard function creation, but also adds the name and a self-reference to the function to the function's scope record. This permits recursive functions to be created. *Note:* the name of the function is not added to the current scope, it is permitted in the actual language, but is considered to be syntactic sugar, combining creation and assignment.

(Function call)

$$\frac{\begin{array}{l} H, L, \mathtt{e_1} \longrightarrow H_1, r_1 \qquad \mathsf{This}(H_1, r_1) = l_2 \qquad \gamma(H_1, r_1) = l_1 \\ H_1(l_1, @body) = \lambda \mathtt{x}.\mathtt{e_3} \qquad H_1(l_1, @scope) = L' \\ H_1, L, \mathtt{e_2} \xrightarrow{\gamma} H_2, v \\ H_3 = H_2 \uplus \mathsf{act}(l, \mathtt{x}, v, \mathtt{e_3}, l_2) \\ H_3, l{:}L', \mathtt{e_3} \xrightarrow{\gamma} H', v' \end{array}}{H, L, \mathtt{e_1}(\mathtt{e_2}) \longrightarrow H', v'}$$

Executes the body of the function, using the passed expression as the value to bind to the parameter. The body of the function is executed in the scope stored with the function, any variables defined within the function body are defined on an activation record so that they are lexically scoped.

(Object construction)

$$\frac{\begin{array}{l} H, L, \mathtt{e_1} \xrightarrow{\gamma} H_1, l_1 \qquad l_1 \neq \mathtt{null} \qquad H_1(l_1, @body) = \lambda \mathtt{x.e_3} \\ H_1(l_1, @scope) = L' \qquad H_1(l_1, \mathtt{prototype}) = v \\ H_1, L, \mathtt{e_2} \xrightarrow{\gamma} H_2, v_1 \qquad l_2 = \mathsf{SelectProto}(v) \\ H_3 = H_2 \uplus \mathsf{obj}(l_3, l_2) \uplus \mathsf{act}(l, \mathtt{x}, v_1, \mathtt{e_3}, l_3) \\ H_3, l{:}L', \mathtt{e_3} \xrightarrow{\gamma} H', v_2 \qquad \mathsf{getBase}(l_3, v_2) = l' \end{array}}{H, L, \mathtt{new\ e_1(e_2)} \longrightarrow H', l'}$$

Constructs an object using the given function, an object is created as usual, it's prototype is assigned to that of the function's `prototype` field. The body of the function is then executed, commonly used to initialize the newly created object.

(This)

$$\frac{\begin{array}{l} \sigma(H, L, @this) = l \\ (l, @this) \mapsto l' \end{array}}{H, L, \mathtt{this} \longrightarrow H, l'}$$

The `this` expression is context-dependent. When used outside of a function, it should evaluate to the most global accessible scope. When used within a function, if the function is called using Member Access (such as `ob.f()`), it will evaluate to the object on which that function was called (in this case: `ob`). Otherwise (most likely a direct function call such as `f()`), `this` returns undefined. Note that this is particularly problematic because functions can easily be aliased off of the object on which they appear to be defined. For this reason the code `g = ob.f; ob.f()` may behave significantly differently from the code `g = ob.f; g()`

These semantics match ES5.1 Strict mode semantics and also agree with the JS-based implementation of the SES language.

3.2.4 SES-specific Expressions

(Restricted evaluation)

$$\frac{\begin{array}{l} H, L, \mathtt{e_1} \xrightarrow{\gamma} H_1, v \qquad \mathtt{e_3} = \mathsf{parse}(v) \\ H_1, L, \mathtt{e_2} \xrightarrow{\gamma} H_2, l \\ H_3 = H_2 \uplus l' \mapsto \{@this : l, @proto : \mathtt{null}\} \uplus \mathsf{defs}(\_, l', \mathtt{e_3}) \\ H_3, l'{:}[l], \mathtt{e_3} \xrightarrow{\gamma} H', v \end{array}}{H, L, \mathtt{reval(e_1, e_2)} \longrightarrow H', v}$$

Parse the first given expression as SES code. A new scope chain is prepared. Its root is the object specified by the second parameter, the *imports* to the restricted environment. An activation record which initializes any variables declared in the parsed source is then appended. This may cause imported objects to be shadowed. The return value of the restricted evaluation statement is the same as that of the final statement of the source to be executed.

(Freeze)

$$\frac{\begin{array}{l} H, L, \mathtt{e} \xrightarrow{\gamma} H'', l \\ H' = H''[(l, @frozen) \mapsto \mathtt{true}] \end{array}}{H, L, \mathtt{freeze(e)} \longrightarrow H', l}$$

Makes the provided object read-only. This prevents field additions, modifications and deletions.

This command is not present in the previous JS program logic. It is a primitive specified in the ES5 spec, however this spec has a bug, meaning that frozen fields on an object's prototype are not permitted to be overridden. Most browsers opt to "fix" the spec bug, however, we follow the ES5 spec in this case.

(Recursive freeze)

$$\frac{H, L, \mathtt{e} \xrightarrow{\gamma} H'', l \quad H' = H''[\mathsf{auxDef}(H'', l, \{\})]}{H, L, \mathtt{def}(\mathtt{e}) \longrightarrow H', l}$$

Recursively calls freeze on all objects reachable via user-defined fields from the given object.

We use a new auxiliary function, auxDef here to handle the recursion onto fields of the object, whilst avoiding non-termination due to cyclical object graphs. It is defined as:

$$\mathsf{auxDef}(H, l, s) \triangleq \begin{cases} \mathtt{emp} & l \in s \vee l \notin \mathcal{L} \\ (l, @frozen) \mapsto \mathtt{true} \cup \bigcup_{(l,x_n) \in H, x_n \in \mathcal{X}^{\upsilon}} \mathsf{auxDef}(H, H(l, x_n), s \cup \{l\}) & \text{otherwise} \end{cases}$$

The Def command is defined in SES as a function that uses Freeze and the `for..in` syntax to loop over fields on the object. We have not chosen to model this syntax for technical reasons[1].

[1] The ES specification for `for..in` is non-deterministic! (Even between two consecutive executions of the same loop under the same conditions!)

Chapter 4

Program Logics

Although the operational semantics provide a precise description of how a program will execute, their use for practical program verification is long-winded, tedious and error-prone. Instead, we use a program logic, which allows us to reason at the level of abstraction best suited to the particular program we would like to verify. Since so much of the behaviour of JS and SES is dependant on the state of the program heap, it is natural to start with separation logic.

We first extend separation logic to be able to express not only what can be accessed *from* a given object, but also what paths exist to obtain access *to* a given object. We then produce a Hoare-style inference rule for each expression in the syntax of the language.

4.1 Extending Separation Logic

As previously discussed in section 2.3.2, SES is an object capability language where references to objects are not forgeable. To reason about the security properties of SES programs, we need to be able to assert that only trusted objects contain references to sensitive data or functionality. Separation logic is unable to make statements like this, since the statement is inherently *non-local*. If we attempt to make such a statement, then at any time, we might use the frame rule to introduce an object which contradicts the statement we made.

One naive method to solve this would be to expand the footprint of instructions to cover all the portions of the heap that could potentially point to the object we wish to check is secure. However, this would quickly cause separation logic to lose its advantages over normal boolean logics – the pre- and post-conditions for rules now no longer precisely state which portions of the heap are required for the command to function.

We instead extend the logic to support these sorts of statements by: reintroducing a global heap state to the assertion satisfaction relation; redefining the existing logical operators to carry around the global heap state without examining it; and introducing new logical operators that allow us to make limited non-local assertions without breaking the frame rule.

4.1.1 Intuition

Regular separation logic for JS assertions are defined using a *satisfaction relation* of the following form:

$$h, L, \epsilon \models P \iff \text{some conditions}$$

In this relation: h is the portion of the JS heap currently under consideration; L is a list of pointers to objects in the heap which are currently serving as an emulated variable store; and ϵ is a logical environment which defines the meanings of the logical variables used in the assertion P.

In order to allow limited assertions about the global heap, we must define our assertions slightly differently. We use a satisfaction relation of the following form:

$$h, h_g, L, \epsilon \models P \iff \text{some conditions}$$

The one change is the addition of the *global heap* h_g. We require that h must always be a sub-heap of h_g. That is to say: the portion of the heap under consideration must be a part of the whole heap. In the following sections we describe how this change threads through the definitions of the existing regular separation logic assertions, and how it allows us to create new sorts of assertion.

4.1.2 Existing Separation Logic Operators

The pure assertions, $\text{true}, \text{false}, \wedge, \vee, \neg$ extend trivially with the addition of the new heap state, for example,the new definition of $\wedge$ is:

$$h, h_g, L, \epsilon \models P \wedge Q \iff (h, h_g, L, \epsilon \models P) \wedge (h, h_g, L, \epsilon \models Q)$$

The definitions of the separating conjunction $*$ and partially separating conjunction $\between$ also simply extend by passing through the global heap state:

$$h, h_g, L, \epsilon \models P * Q \iff \exists h_1, h_2. h \equiv h_1 \uplus h_2 \wedge (h_1, h_g, L, \epsilon \models P) \wedge (h_2, h_g, L, \epsilon \models Q)$$
$$h, h_g, L, \epsilon \models P \between Q \iff \exists h_1, h_2, h_3. h \equiv h_1 \uplus h_2 \uplus h_3 \wedge (h_1 \uplus h_3, h_g, L, \epsilon \models P) \wedge (h_2 \uplus h_3, h_g, L, \epsilon \models Q)$$

4.1.3 Magic Wand

The separating implication, or magic wand, operator, $-\!\!*$, is usually defined as follows:

$$h \models P -\!\!* Q \iff \forall h'. (h' \models P) \wedge h \# h' \Rightarrow (h \uplus h' \models Q)$$

It means that if the heap is extended by a disjoint portion that satisfies P, then the resulting heap will satisfy Q.

The magic wand is generally used for the generation of weakest preconditions of a command. Our extension to Separation Logic shows that there are two cases of weakest precondition generation.

The first of these is when the weakest precondition is extending the heap footprint with information consistent with the heap, for example the weakest precondition for object creation of the form $P -\!\!* Q$ would have the object asserted in Q, and the footprint of the subexpressions used to populate the object's fields in P.

The second case is when the weakest precondition is used for speculating or *hypothesising* new values in the existing heap. For example, in the weakest precondition for the overwriting assignment.

The magic wand operator is traditionally defined to be the right adjoint of $*$:

$$P * Q \vdash R \text{ iff } P \vdash Q -\!\!* R$$

Since separation logic usually defines $-\!\!*$ as follows:

$$h \models P -\!\!* Q \iff \forall h'. (h' \models P) \wedge h \# h' \Rightarrow (h \uplus h' \models Q)$$

We might define $-\!\!*$ in our context to preserve the right adjoint property like so:

$$h, h_g \models P -\!\!* Q \iff \forall h'.\, (h', h_g \models P) \wedge h \# h' \Rightarrow (h \uplus h', h_g \models Q)$$

Now consider $x \mapsto 1 * (x \mapsto 2 -\!\!* Q)$, an instance of the hypothesising weakest precondition noted earlier.

In regular separation logic, this assertion is satisfiable. Consider the heap $x : 1$ as follows:

$$\begin{aligned} & x : 1 \models x \mapsto 1 * (x \mapsto 2 -\!\!* Q) \\ \text{iff } & x : 1 \models x \mapsto 1 \wedge \oslash \models (x \mapsto 2 -\!\!* Q) \\ \text{iff } & x : 1 \models x \mapsto 1 \wedge x : 2 \models x \mapsto 2 \wedge (x : 2 \uplus \oslash) \models Q \end{aligned}$$

But with our naive definition of $-\!\!*$, this assertion is *unsatisfiable*. Consider again the heap $x : 1$, which satisfied the regular separation logic version of this assertion:

$$\begin{aligned} & x : 1, x : 1 \models x \mapsto 1 * (x \mapsto 2 -\!\!* Q) \\ \text{iff } & x : 1, x : 1 \models x \mapsto 1 \wedge \oslash, x : 1 \models (x \mapsto 2 -\!\!* Q) \\ \text{iff } & x : 1, x : 1 \models x \mapsto 1 \wedge x : 2, x : 1 \models x \mapsto 2 \wedge (x : 2 \uplus \oslash, x : 1) \models Q \end{aligned}$$

Note that the local and global heap states do not agree in the derivation of the sub-assertions. But the local portion of the heap should always be a subset of the global portion! Since we cannot find local and global heaps for these sub-derivations which agree, the heap $x : 1$ does not satisfy our original assertion.

This behaviour prevents us from using $-\!\!*$ for hypothetical heap update operations. To support this use-case, we introduce a new logical operator.

The new operator to be defined is the *box-wand*, $-\!\!\circledast$ or "hypothesising separating implication". In addition to the behaviour of the standard wand, the box-wand maintains consistency between the local and global heap states whilst evaluating the satisfaction of sub-assertions of the operator. This permits a state to be hypothesised that conflicts with the current global state of the heap.

$$h, h_g \models P -\!\!\circledast Q \iff \forall h'.\, (h', h_g[h'] \models P) \wedge h \# h' \Rightarrow (h \uplus h', h_g[h'] \models Q)$$

The derivation our example now behaves as expected, and the assertion is satisfied.

$$\begin{aligned} & x : 1, x : 1 \models x \mapsto 1 * (x \mapsto 2 -\!\!\circledast Q) \\ \text{iff } & x : 1, x : 1 \models x \mapsto 1 \wedge \oslash, x : 1 \models (x \mapsto 2 -\!\!\circledast Q) \\ \text{iff } & x : 1, x : 1 \models x \mapsto 1 \wedge x : 2, x : 2 \models x \mapsto 2 \wedge (x : 2 \uplus \oslash, x : 2) \models Q \end{aligned}$$

4.1.4 Backpointer

We now define the operator that will primarily examine the global heap state, the *backpointer*, $E_1 \leftarrow E_2$, which specifies that any heap cell that has the E_1 as its value must appear in the set E_2. The set E_2 may additionally contain extra heap cell references. This allows us to express the permission that a given cell may be changed to point to E_1 in the future.

The assertion $E_1 \leftarrow E_2$ means that the location E_1 may be pointed to by *at most* the locations in E_2.

$$\begin{aligned} h, h_g \models E_1 \leftarrow E_2 \iff & \forall (l, x) \in \mathsf{dom}(h_g).\, h_g(l, x) = [\![E_1]\!]^L_\epsilon \Rightarrow (l, x) \in [\![E_2]\!]^L_\epsilon \wedge \\ & h \equiv ([\![E_1]\!]^L_\epsilon, @bp) \mapsto \_ \end{aligned}$$

Note that the interaction between the backpointer operator and the consequence rule in the Hoare reasoning is particularly useful. For example, we can widen the backpointer set at any time necessary thanks to the implication $l \leftarrow S \Rightarrow l \leftarrow S \cup S'$.

We are also able to shrink the set if it is known that the elements being removed definitely do not point to the object in question:

$$a \leftarrow \{(b,c)\} * (b,c) \mapsto d * a \neq d \Rightarrow a \leftarrow \{\} * (b,c) \mapsto d$$

Although the backpointer operator would at first seem to not require any footprint on the heap (that it would be a *pure* operator), the operator's interactions with the spatial elements of the logic may result in unsoundness.

For example, consider the following *broken* proof outline:

$\{\mathtt{y} \mapsto \_ * \mathtt{x} \mapsto a \wedge a \leftarrow \{\mathtt{x}\}\}$
[Consequence]
$\{(\mathtt{y} \mapsto \_ * \mathtt{x} \mapsto a \wedge a \leftarrow \{\mathtt{x}\}) * (a \leftarrow \{\mathtt{x}\} \wedge \mathtt{emp})\}$
[Frame off]
$\{\mathtt{y} \mapsto \_ * \mathtt{x} \mapsto a \wedge a \leftarrow \{\mathtt{x}\}\}$
[Consequence]
$\{\mathtt{y} \mapsto \_ * \mathtt{x} \mapsto a \wedge a \leftarrow \{\mathtt{x},\mathtt{y}\}\}$
$\mathtt{y} = \mathtt{x};$
$\{\mathtt{y} \mapsto a * \mathtt{x} \mapsto a \wedge a \leftarrow \{\mathtt{x},\mathtt{y}\}\}$
[Frame on]
$\{(\mathtt{y} \mapsto a * \mathtt{x} \mapsto a \wedge a \leftarrow \{\mathtt{x},\mathtt{y}\}) * (a \leftarrow \{\mathtt{x}\} \wedge \mathtt{emp})\}$
[Consequence]
$\{\mathtt{y} \mapsto a * \mathtt{x} \mapsto a \wedge a \leftarrow \{\mathtt{x}\}\}$

Figure 4.1: Broken proof showing requirement for $\leftarrow$ to have a footprint

Notice that in this *broken* program proof, we appear to have proved that our program ensures that both y and x point to a and *also* that the only location pointing to a is x. To prevent these sorts of shenanigans, we insist that two backpointer assertions about the same location cannot be separated. The mechanism we use to enforce this requirement is a notional internal field $@bp$ on each SES object. This field is the footprint of any backpointer assertions about that object.

4.2 Hoare Triples

Recall Section 2.2.2 where Hoare reasoning was introduced, in this section, we extend Hoare reasoning to support our extensions to separation logic and we define inference rules for the SES language.

4.2.1 Definition

We define the syntax $\{P\}\mathtt{e}\{Q\}$ to be a Hoare triple, where P is the precondition, e is the subexpression, and Q is the postcondition that results from evaluating e in a state that satisfies P.

It is necessary that we relate the Hoare triples to the Operational Semantics of the language defined earlier.

The first of these properties is the soundness property, it specifies that if the command e is evaluated in a heap h, such that h is the footprint of and satisfies the expression's precondition P; and it is a

subheap of the entire heap $h \uplus h_f$, then the evaluation will not fault and if it terminates, will terminate with a subheap h' that satisfies the expression's postcondition Q. Or, more simply:

$$(h, h \uplus h_f, L, (\epsilon \setminus \mathbf{r}) \models P) \wedge h, L, \mathsf{e} \rightsquigarrow h', v \Rightarrow (h', h' \uplus h_f, L, [\epsilon | \mathbf{r} \leftarrow v] \models Q)$$

As we are using the fault-avoiding Hoare triple semantics, the expression must not fault if it is evaluated in a state that satisfies the triple's precondition. We must also define a fault avoidance property to be able to verify that this is the case:

$$h, h_g, L, (\epsilon \setminus \mathbf{r}) \models P \wedge h \subseteq h_g \Rightarrow h, L, \mathsf{e} \not\rightsquigarrow \mathsf{fault}$$

We also define the Safety Monotonicity and Frame properties, which guarantee that the operational semantics and separation logic reasoning framework are consistent with regard to the spatial reasoning about the heap.

Safety Monotonicity states that if an expression evaluates without fault from a state that satisfies the precondition, then disjointly extending the heap will not cause the expression to fault.

$$(h, h_g, L, \epsilon \models P) \wedge h \# h' \wedge h, L, \mathsf{e} \not\rightsquigarrow \mathsf{fault} \wedge h \subseteq h_g \Rightarrow h \uplus h', L, \mathsf{e} \not\rightsquigarrow \mathsf{fault}$$

The Frame property states that if an expression is evaluated from a state with heap h without faulting to produce h_2, then the same expression is evaluated from the same state but with the heap $h \uplus h'$, to produce the heap h'_2, then the second evaluation will not have touched any part of h', namely that $h'_2 = h_2 \wedge h'$:

$$(h, h_g, L, \epsilon \models P) \wedge h, L, \mathsf{e} \not\rightsquigarrow \mathsf{fault} \wedge h \uplus h', L, \mathsf{e} \rightsquigarrow h'_2 \wedge h \subseteq h_g \Rightarrow h, L, \mathsf{e} \rightsquigarrow h_2 \wedge h'_2 = h_2 \uplus h'$$

Ideally, we would prove that each of these properties hold for each pair of the operational semantics and program logic rules. These results generally provide little new intuition about the semantics or logic, so this exercise has been omitted from this project, in favour of more targeted real-world examples.

4.2.2 Inference Rules

Inference rules for Hoare triples are similar in nature to those for operational semantics. Derivation of the pre- and post-conditions for the Hoare triple of a given expression usually requires the recursive derivation of its subexpressions, along with some structural operations on their intermediary assertions to build the overall pre- and post-conditions.

The compositional nature (and the power of the Frame rule) of Separation logic is clear from the structure of the inference rules, only the information required to prove the particular operation is required – all assertions required to satisfy subexpressions are transparently passed through to them and assertions that are not required for a particular sub-derivation can be hidden by the frame rule.

The benefits of separation logic over the operational semantics are also clear here – the pre- and post-conditions of a given expression will state clearly what the expression will access.

As with the operational semantics section of the project, the Inference Rules for the SES language are necessarily based upon the JS inference rules produced in [13]. However, here many more adaptions were required to introduce the backpointer assertions required by the logic to track assignments.

It is useful to note that the Operational Semantics rules need not necessarily map to Hoare triple inference rules. Indeed, in the case of the while statement, the two operational semantics rules for the true/false cases are mapped down to the single Hoare triple inference rule.

All of the inference rules for Structural Expressions listed in Section 3.2.1 and the standard Separation Logic axioms of Frame, Elimination, Consequence and Disjunction are used unmodified.

I will discuss the Binary Operator inference rule in detail to explain how to interpret some of the techniques used in the these rules.

How to Read Hoare Triple Inference Rules

(Binary Operator)

$$\frac{\begin{array}{l}\{P\}\mathtt{e_1}\{R * \mathbf{r} \doteq V_1\} \quad R = S_1 * \gamma(Ls_1, V_1, V_3) \\ \{R\}\mathtt{e_2}\{Q * \mathbf{r} \doteq V_2\} \quad Q = S_2 * \gamma(Ls_2, V_2, V_4) \\ V = V_3 \mathbin{\bar{\oplus}} V_4\end{array}}{\{P\}\mathtt{e_1} \oplus \mathtt{e_2}\{Q * \mathbf{r} \doteq V\}}$$

It is first useful to note the overall 'flow' of the assertions, here, P is the overall precondition as well as the precondition to $\mathtt{e_1}$, whose post-condition is the pre-condition to $\mathtt{e_2}$, whose post-condition is also that of the entire expression.

We next look at the return value for each sub-expression, $\mathbf{r}$. It is presented as being extra to Q and R, both so that its value may be used, and so that is is not passed into the pre-condition of the next sub-expression.

The structure of the intermediary assertion R is stated as having the form $S_1 * \gamma(Ls_1, V_1, V_3)$, which has the meaning 'get value', it must be a part of R, since the derivation for $\mathtt{e_1}$ may use it (or even produce it) and, if not, it will be passed through from P by the frame rule.

The γ predicate has been lifted to the logic from the semantics, the definition is unchanged from [13] and is given in Appendix C.

Finally, from the retrieved values, V_3 and V_4, V is calculated using the operator $\bar{\oplus}$ (lifted to the logic), and the value set as the return value for the expression.

Heap-based Expressions

(Variable Resolution)

$$\frac{P = \sigma(Ls_1, \mathbf{l}, \mathtt{x}, L) \between \gamma(Ls_2, L{\cdot}\mathtt{x}, V) * L \neq \mathtt{null}}{\{P\}\mathtt{x}\{P * \mathbf{r} \doteq L{\cdot}\mathtt{x}\}}$$

This rule searches the current scope chain, $\mathbf{l}$ for the variable $\mathtt{x}$, placing all searched heap cells into the assertion. In addition to the counterpart operational semantics rule, permission to get the value of the found variable must also be specified as *partially* disjoint to the scope chain search, as both make reference to the same target heap cell.

The returned reference is required to be not on the `null` object by the SES language, this requirement is not enforced by JS, as discussed in Section 3.2.2.

(Member access)

$$\frac{\{P\}\mathtt{e}\{Q * \mathbf{r} \doteq V\} \quad Q = R * \gamma(Ls, V, L) * L \neq \mathtt{null} * L \doteq \mathcal{L}}{\{P\}\mathtt{e.x}\{Q * \mathbf{r} \doteq L{\cdot}\mathtt{x}\}}$$

(Computed member access)

$$\frac{\begin{array}{l}\{P\}\mathtt{e_1}\{R * \mathbf{r} \doteq V_1\} \quad R = S_1 * \gamma(Ls_1, V_1, L) * L \neq \mathtt{null} * L \doteq \mathcal{L} \\ \{R\}\mathtt{e_2}\{Q * X \doteq \mathcal{X}^{\mathrm{U}} * \mathbf{r} \doteq V_2\} \quad Q = S_2 * \gamma(Ls_2, V_2, X)\end{array}}{\{P\}\mathtt{e_1}[\mathtt{e_2}]\{Q * \mathbf{r} \doteq L{\cdot}X\}}$$

Member access and Computed member access rules are the same as for JS. The checks for value type being a location or a user-definable field ($L \dot{\in} \mathcal{L}$ and $X \dot{\in} \mathcal{X}^{\mathrm{U}}$) are implicit in the operational semantics, but explicit in the logic.

(Assign)

$$\frac{\begin{array}{l}\{P\}\mathtt{e_1}\{R * \mathbf{r} \doteq L{\cdot}X\} \\ \{R\}\mathtt{e_2}\{Q * (L, X) \mapsto V_3 * \beta(V_2, s) * \mathbf{r} \doteq V_1\} \\ Q = S * \gamma(Ls, V_1, V_2) * \mathsf{RW}(L)\end{array}}{\{P\}\mathtt{e_1} = \mathtt{e_2}\{Q * (L, X) \mapsto V_2 * \beta(V_2, L, X, s) * \mathbf{r} \doteq V_2\}}$$

The assignment rule has been considerably simplified as a result of SES not permitting assignments to undeclared variables. JS had two rules for this case, one to handle assignment to the global object in the case where (`null`, X) was the target field, the other to handle the regular assignment, as shown above.

To support the new SES `freeze()` command, we introduce a check that the object being assigned to is readable using the RW predicate:

$$\mathsf{RW}(L) \triangleq (L, @frozen) \mapsto \mathtt{false}$$

Where it does not harm readability, it is good to have inference rules that are as general as possible. It is often the case in the inference rules that we do not know what type will be present in a heap cell. The $\leftarrow$ operator is defined over only heap locations, and not pure values, so it would be incorrect to use the $\leftarrow$ operator with any value that we cannot guarantee must be a location. Given that it is arduous to introduce multiple inference rules to handle these cases (and impossible when an unknown number of subexpressions are being handled), we instead introduce the β predicate, which has the purpose of being a type-agnostic version of the $\leftarrow$ operator:

$$\begin{aligned}\beta(V, \_) &\triangleq V \dot{\notin} \mathcal{L} \\ \beta(V, s) &\triangleq V \dot{\in} \mathcal{L} * V \leftarrow s\end{aligned}$$

We additionally also define a 4-ary version to simplify the oft-used extension of the backpointer set s by the heap cell (L, x):

$$\begin{aligned}\beta(V, \_, \_, \_) &\triangleq V \dot{\notin} \mathcal{L} \\ \beta(V, L, x, s) &\triangleq V \dot{\in} \mathcal{L} * V \leftarrow \{(L, x)\} \cup s\end{aligned}$$

Note that the assign rule could also be written with a single, 4-ary instance of β as part of the definition of Q This alternative definition relies on the backpointer set widening consequence/implication as discussed in Section 4.1.4. However, it is the author's preference to give the explicit widenings in the rule definition.

(Object creation)

$$\frac{\begin{array}{l}\forall i \in 1..n. \left(\begin{array}{l} P_i = R_i * \gamma(Ls_i, Y_i, X_i) * \beta(X_i, s_i) \\ \{P_{i-1}\}\mathtt{e_i}\{P_i * \mathbf{r} \doteq Y_i\}\end{array}\right) \\ P_n = R * l_{op} \leftarrow s_{op} \\ Q = R * \exists L. \left(\begin{array}{l} \mathsf{newobj}_L(@proto, \mathtt{x_1}, ..., \mathtt{x_n}) * \\ \circledast_{1 \le i \le n}((L, \mathtt{x_i}) \mapsto X_i * \beta(X_i, L, \mathtt{x_i}, s_i)) * \\ (L, @proto) \mapsto l_{op} * l_{op} \leftarrow s_{op} \cup \{(L, @proto)\} * \\ \mathbf{r} \doteq L * L \leftarrow \{\}\end{array}\right) \\ \mathtt{x_1} \neq \cdots \neq \mathtt{x_n} \qquad \mathbf{r} \notin \mathsf{fv}(P_n)\end{array}}{\{P_0\}\{\mathtt{x_1} : \mathtt{e_1}, ..., \mathtt{x_n} : \mathtt{e_n}\}\{Q\}}$$

This rule traverses the sub-expressions in the construction in left-to-right order, collecting the required pre-conditions and post-conditions to satisfy all expressions. For each sub-expression, the return value and backpointer set for each object that is returned is stored for the assignment.

The object is then created, assigning the values stored in the first pass, and updating their backpointer sets as necessary. The prototype of the object is set to the language constant l_{op} (the object prototype) and its backpointer set updated appropriately. Finally, a backpointer assertion for the newly created object is produced for the new object.

This rule is sound, but is known not to be complete – it fails for the creation of objects which assign the same location to multiple fields. Further work is needed to fix this.

Function-related Expressions

(Function)

$$\frac{P = l_{op} \leftarrow s_1 * l_{fp} \leftarrow s_2 * \mathsf{scopeBps}(\mathbf{l}, ss) \qquad Q = \exists L_1, L_2. \left(\begin{array}{l} \mathsf{fullobj}_{L_1}(@proto : l_{op}) * L_1 \leftarrow \{(L_2, \texttt{prototype})\} * l_{op} \leftarrow s_1 \cup \{(L_1, @proto)\} * \\ \mathsf{newfun}_{L_2}(\mathbf{l}, \texttt{x}, \texttt{e}, L_1) * L_2 \leftarrow \{\} * l_{fp} \leftarrow s_2 \cup \{(L_2, @proto)\} * \\ \mathbf{r} \doteq L_2 * \mathsf{scopeBpsUpd}(\mathbf{l}, ss, ss', \{(L_2, @scope)\}) \end{array} \right)}{\{P\}\texttt{function (x)\{e\}}\{Q\}}$$

(Named Function)

$$\frac{P = l_{op} \leftarrow s_1 * l_{fp} \leftarrow s_2 * \mathsf{scopeBps}(\mathbf{l}, ss) \qquad Q = \exists L_1, L_2, L_3. \left(\begin{array}{l} \mathsf{fullobj}_{L_1}(@proto : l_{op}) * L_1 \leftarrow \{(L_2, \texttt{prototype})\} * l_{op} \leftarrow s_1 \cup \{(L_1, @proto)\} * \\ \mathsf{newfun}_{L_2}((L_3 : \mathbf{l}), \texttt{x}, \texttt{e}, L_1) * L_2 \leftarrow \{(L_3, \texttt{y})\} * l_{fp} \leftarrow s_2 \cup \{(L_2, @proto)\} * \\ \mathsf{fullobj}_{L_3}(@proto : \texttt{null}, \texttt{y} : L_2) * L_3 \leftarrow \{(L_2, @scope)\} * \\ \mathbf{r} \doteq L_2 * \mathsf{scopeBpsUpd}(\mathbf{l}, ss, ss', \{(L_2, @scope)\}) \end{array} \right)}{\{P\}\texttt{function y(x)\{e\}}\{Q\}}$$

Prior to the extension of the logic, both these functions required just $\oslash$ for their preconditions, they now require 3 backpointer assertions, the first two are for the language constants l_{op} and l_{fp} (the object and function prototypes). The third of these assertions, scopeBps fetches the backpointer sets for each of the objects in the scope chain, $\mathbf{l}$, and places them in the list ss of backpointer sets.

The post-conditions for these commands construct 2 or 3 objects: the function itself (L_2); a prototype object (L_1); and, for the named variant, a scope record (L_3) binding the function's name to itself for use in recursive calls. Fresh backpointer assertions are made for each of these new objects.

Since functions store the scope in which they are created the set of objects in the scope chain also require their backpointer sets updating. This is because the body of the function, and thus the function itself now potentially provides a means of accessing those objects. The scopeBpsUpd predicate uses the ss list stored by the scopeBps predicate in the precondition to achieve this.

$$\mathsf{scopeBpsUpd}(Ls, ss, ss', n) \triangleq \circledast_{0 \leq i < \mathrm{length}(Ls)} (\mathrm{item}(i, Ls) \leftarrow \mathrm{item}(i, ss') * \mathrm{item}(i, ss') \doteq \mathrm{item}(i, ss) \cup n)$$

$$\mathsf{scopeBps}(Ls, ss) \triangleq \mathsf{scopeBpsUpd}(Ls, ss, ss, \{\})$$

The following equivalence for the scopeBpsUpd predicate is useful to note for use in proofs:

$$\mathsf{scopeBpsUpd}(L : Ls, s : ss, (s \cup n) : ss', n) \iff L \leftarrow s \cup n * \mathsf{scopeBpsUpd}(Ls, ss, ss', n)$$

(Function Call)

$$\frac{\begin{array}{l}\{P\}\mathtt{e_1}\{R_1 * \mathbf{r} \doteq F_1\} \\ R_1 = \left(\begin{array}{l} S_1 \uplus \mathsf{This}(F_1, T) \uplus \gamma(Ls_1, F_1, F_2) * \\ (F_2, @body) \mapsto \lambda X.\mathtt{e_3} * (F_2, @scope) \mapsto Ls_2 \end{array}\right) \\ \{R_1\}\mathtt{e_2}\{R_2 * \beta(T, s) * \beta(V_2, s') * \mathbf{l} \doteq Ls_3 * \mathbf{r} \doteq V_1\} \\ R_2 = S_2 * \gamma(Ls_4, V_1, V_2) \\ R_3 = R_2 * \exists L. \left(\begin{array}{l} (L, X) \mapsto V_2 * \beta(V_2, L, X, s') * \\ (L, @this) \mapsto T * \beta(T, L, @this, s) * (L, @proto) \mapsto \mathtt{null} * \\ \mathsf{defs}(X, L, \mathtt{e_3}) * \mathsf{newobj}_L(@proto, @this, \mathtt{x}, \mathsf{decls}(X, L, \mathtt{e_3})) * \\ L \leftarrow \{\} * \mathbf{l} \doteq L : Ls_2 \end{array}\right) \\ \{R_3\}\mathtt{e_3}\{\exists L.\, Q * \mathbf{l} \doteq L : Ls_2\} \qquad \mathbf{l} \notin \mathsf{fv}(Q) \cup \mathsf{fv}(R_2) \end{array}}{\{P\}\mathtt{e_1}(\mathtt{e_2})\{\exists L.\, Q * \mathbf{l} \doteq Ls_3\}}$$

The function call rule is largely unchanged from the JS version, changes are made to introduce back-pointer set updates for the object assigned to the @this internal variable, the parameter of the function, and created for the activation record object.

(This)

$$\frac{P = \sigma(Ls_1, \mathbf{l}, @this, L_1) \uplus \pi(Ls_2, L_1, @this, L_2) \uplus (L_2, @this) \mapsto V}{\{P\}\mathtt{this}\{P * \mathbf{r} \doteq V\}}$$

The This rule is the same as in the JS language. The object that this is bound to is determined by the value of the internal @this variable, set on function activation records and the object at the bottom of the scope chain.

SES-specific Expressions

(Restricted evaluation)

$$\frac{\begin{array}{l}\{P\}\mathtt{e_1}\{R_1 * \mathbf{r} \doteq V_1\} \\ R_1 = S_1 * \gamma(\_, V_1, V_2) * V_2 \dot{\in} \mathcal{M} \\ \mathsf{parse}(V_2) = \mathtt{e_3} \\ \{R_1\}\mathtt{e_2}\{R_2 * \mathbf{r} \doteq V_3 * \mathbf{l} \doteq Ls\} \\ R_2 = \left(\begin{array}{l} S_2 * \gamma(\_, V_3, V_4) * V_4 \dot{\in} \mathcal{L} * V_4 \leftarrow s \cup \{(L, @this)\} * \\ \exists L.\, R_3 * \mathsf{newobj}_L(@proto, @this, \mathsf{decls}(\_, L, \mathtt{e_3})) * \\ \mathsf{obj}_L(@this : V_4, @proto : \mathtt{null}) * \mathsf{defs}(\_, L, \mathtt{e_3}) \end{array}\right) \\ R_3 = (\mathbf{l} \doteq L : [V_4]) \\ \{R_2\}\mathtt{e_3}\{\exists L.\, Q * R_3\} \end{array}}{\{P\}\mathtt{reval}(\mathtt{e_1}, \mathtt{e_2})\{\exists L.\, Q * \mathbf{l} \doteq Ls\}}$$

This is a new definition. The first subexpression, the untrusted code, is resolved to a string which is parsed into a suitable form for analysis (details out of scope of this formalisation). The second subexpression is resolved to an object, which is used as the base object for a new scope chain. The untrusted code is then executed within the context of the new scope chain. Its pre- and post-conditions are incorporated into those of the whole expression.

We assume that a Hoare triple for the untrusted code is available, or derivable. If not, an arbitrary triple could be substituted, say one that is designed to be *maximally malicious*. If the security of the restricted evaluation of this program can be proven, then all programs run in this context should be safe.

It is the case that the object resolved by the second subexpression of a call to restricted evaluation should be considered security critical. It is reachability from this object to privileged objects that should be checked by axioms that verify security properties.

(Freeze)

$$\frac{\{P\}\mathtt{e}\{Q * \mathbf{r} \doteq V_1 * (V_2, @frozen) \mapsto V_3\} \quad Q = \gamma(Ls, V_1, V_2) * S}{\{P\}\mathtt{freeze(e)}\{Q * (V_2, @frozen) \mapsto \mathtt{true} * \mathbf{r} \doteq V_2\}}$$

The rule for Freeze is straightforward, it simply requires permission to the @frozen internal field of the object being frozen, and asserts that it is set to true in the postcondition.

(Recursive freeze)

$$\frac{\{P\}\mathtt{e}\{Q * \mathbf{r} \doteq V_2 * \mathsf{auxDefGet}(V_2, \{\})\} \quad Q = \gamma(Ls, V_1, V_2) * S}{\{P\}\mathtt{def(e)}\{Q * \mathbf{r} \doteq V_2 * \mathsf{auxDefSet}(V_2, \{\}, \mathtt{true})\}}$$

Similar to the freeze expression, the recursive freeze requires permission to access the @frozen field on accessible objects. It achieves this through use of the auxDefGet and auxDefSet predicates:

$$\mathsf{auxDefGet}(V, s) \triangleq V \notin \mathcal{L} \vee V \in s$$

$$\mathsf{auxDefGet}(V, s) \triangleq V \notin s \wedge (V, @frozen) \mapsto \_ * (\underset{x_n \in \mathcal{X}^{\mathrm{U}}}{\lfloor * \rfloor} (V, x_n) \mapsto V' \bowtie \mathsf{auxDefGet}(V', s \cup \{V\}))$$

$$\mathsf{auxDefSet}(V, s, b) \triangleq V \notin \mathcal{L} \vee V \in s$$

$$\mathsf{auxDefSet}(V, s, b) \triangleq V \notin s \wedge (V, @frozen) \mapsto b * (\underset{x_n \in \mathcal{X}^{\mathrm{U}}}{\lfloor * \rfloor} (V, x_n) \mapsto V' \bowtie \mathsf{auxDefSet}(V', s \cup \{V\}, b))$$

These predicates traverse the user-accessible fields of the given object, V, recursively. In the case of the auxDefGet predicate, to assert that the @frozen field is present in the footprint of the precondition, and in the case of auxDefSet that the @frozen field is set to b.

Termination of the predicate is ensured by collecting traversed objects into the set s for the recursive calls, the recursive definition cannot be satisfied if the same location is encountered a second time.

The sepish operator, $\bowtie$, must be used to collect the heap cells into a single assertion. Although we guarantee to terminate on the second traversal of a loop, we must also take into account heap structures that have the same object reachable via different paths, for example the object $\{\mathtt{a} : \mathtt{1}, \mathtt{b} : \mathtt{1}\}$ would be unsatisfiable under the predicates if $*$ were used, since they are not guaranteed to be disjoint. Use of $\bowtie$ is sound in the update case, since the updates are consistent, the same value is written to all the overlapping heap cells.

Chapter 5

Program Proofs

Some design patterns for programming defensively against malicious code of an unknown nature have been developed for use with the Restricted Evaluation command. We have proven the correctness of the security properties of these design patterns using the logic developed in this project.

Both these patterns are designed to wrap security-critical interfaces or objects before being passed into untrusted code. They provide a means of revoking access to those objects when desired by the trusted portion of the code.

Both patterns are capable of being extended to more situation-specific wrappers, for example to deny access to a particular function of an object, or to censor sensitive data as it is passed between contexts.

All of these patterns operate by encapsulating the resources to be protected within the scope of a function. Since the scope is inaccessible by programmatic means from outside the body of the function, the function can be passed to be untrusted code and the reference remain safe. This relies on the correct operation of the function to not disclose the reference by mistake.

These design patterns lie at the boundary between trusted and untrusted code, their verification is critical to guaranteeing security of any program that wishes to share data and privileged functions with untrusted code.

We also demonstrate two styles of proofs, the first demonstrating the logic as being extremely precise with a symbolic execution style of assertion generation, the second showing that the logic is also able to provide more succinct proofs that highlight only the key aspects of evaluation.

5.1 Caretaker

The Caretaker, or Revocable Reference pattern is a rudimentary pattern that demonstrates the ability to encapsulate a reference within a function. The implementation is shown in Figure 5.1.

```
var RevocableRef = function(ref) {
  var protected = ref;
  var access = function(field) { return protected[field]; }
  var kill = function() { protected = null; }
  return {access: access, kill: kill};
}
```

Figure 5.1: Implementation of the Caretaker pattern

When invoked by the trusted code with an object to protect, two functions are returned, an `access` function and a `kill` function. The trusted code keeps the `kill` function to call when the reference is to be revoked. The `access` function is passed to the untrusted code. The `access` function permits fields of the protected object to be accessed until the Caretaker is killed, at which point it must stop functioning.

The only guarantee that the Caretaker pattern provides is that the `access` function will not succeed if the reference has been revoked. The Caretaker pattern provides no guarantees for return values from the protected object, it is therefore not suitable for use with objects that may potentially return bare references into the trusted code. For example, a function that returns itself in response to a particular call would trivially violate the security of the Caretaker.

We have derived assertions for the Caretaker pattern through application of the inference rules to the expressions in a mechanical style (shown in Figures 5.2-5.4). This approach produces a significant number of assertions, of which only a few are relevant to the proof of the trivial security property. Nonetheless, the derivation is a useful demonstration of how the sets of backpointer references are built up by the inference rules.

The derivation proves that once killed, the Caretaker's `access` function will no longer function, the post-condition of `kill` includes $(L, \texttt{protected}) \mapsto \texttt{null}$. The pre-condition of `access` requires that $(L, \texttt{protected}) \mapsto P * P \neq \texttt{null}$. Finally, we note that from the postcondition of `RevocableRef` that the only references to L are $(A, @scope)$ and $(K, @scope)$, namely the externally-inaccessible scope chains of the `access` and `kill` functions. We have already verified that these functions are correct, so the property holds.

$$
\begin{array}{l}
\texttt{var RevocableRef} = \texttt{function(ref)\{} \\
\quad \left\{ \begin{array}{l} \mathsf{obj}_R(@body : \lambda\texttt{ref}.\{\ldots\}, @scope : Ls) * \\ \exists L.\mathbf{l} \doteq L : Ls * \\ \mathsf{fullobj}_L \left(\begin{array}{l} \texttt{ref} : V, @this : \_, @proto : \texttt{null}, @frozen : \texttt{false}, \\ \texttt{protected} : \texttt{undefined}, \texttt{access} : \texttt{undefined}, \texttt{kill} : \texttt{undefined} \end{array} \right) * \\ L \leftarrow \{\} * V \leftarrow s_1 \cup \{(L, \texttt{ref})\} * l_{op} \leftarrow s_3 * l_{fp} \leftarrow s_4 * \mathsf{scopeBps}(Ls, ss) \end{array} \right\} \\
\quad [\text{Frame/elim/cons } (V \leftarrow s_1 \Rightarrow V \leftarrow s_1 \cup s_2)] \\
\quad \left\{ \begin{array}{l} \mathsf{obj}_L(\texttt{protected} : \texttt{undefined}, \texttt{ref} : V, @frozen : \texttt{false}) * \\ V \leftarrow s_1 \cup \{(L, \texttt{ref}), (L, \texttt{protected})\} * \mathbf{l} \doteq L : Ls \end{array} \right\} \\
\quad \texttt{var protected} = \texttt{ref;} \\
\quad \left\{ \begin{array}{l} \mathsf{obj}_L(\texttt{protected} : V, \texttt{ref} : V, @frozen : \texttt{false}) * \\ V \leftarrow s_1 \cup \{(L, \texttt{ref}), (L, \texttt{protected})\} * \mathbf{l} \doteq L : Ls \end{array} \right\} \\
\quad [\text{Frames/cons}] \\
\quad \left\{ \begin{array}{l} \mathsf{obj}_L(\texttt{access} : \texttt{undefined}, @frozen : \texttt{false}) * \\ l_{op} \leftarrow s_3 * l_{fp} \leftarrow s_4 * L \leftarrow \{\} * \mathsf{scopeBps}(Ls, ss) * \mathbf{l} \doteq L : Ls \end{array} \right\} \\
\quad \texttt{var access} = \texttt{function(field)\{}\mathtt{b_a}\texttt{\};} \\
\quad \left\{ \begin{array}{l} \exists A, A_p. \left(\begin{array}{l} \mathsf{obj}_L(\texttt{access} : A, @frozen : \texttt{false}) * A \leftarrow s_7 * s_7 \doteq \{(L, \texttt{access})\} * \\ l_{op} \leftarrow s_5 * s_5 \doteq s_3 \cup \{(A_p, @proto)\} * l_{fp} \leftarrow s_6 * s_6 \doteq s_4 \cup \{(A, @proto)\} * \\ L \leftarrow \{(A, @scope)\} * \mathsf{scopeBpsUpd}(Ls, ss, ss_1, \{(A, @scope)\}) * \\ \mathsf{fullobj}_{A_p}(@proto : l_{op}) * A_p \leftarrow \{(A, \texttt{prototype})\} * \\ \mathsf{newfun}_A(\mathbf{l}, \texttt{field}, b_a, A_p) \end{array} \right) * \mathbf{l} \doteq L : Ls \end{array} \right\} \\
\quad [\text{Frames/cons}] \\
\quad \left\{ \begin{array}{l} \mathsf{obj}_L(\texttt{kill} : \texttt{undefined}, @frozen : \texttt{false}) * \\ l_{op} \leftarrow s_5 * l_{fp} \leftarrow s_6 * L \leftarrow \{(A, @scope)\} * \mathsf{scopeBpsUpd}(Ls, ss_1) * \mathbf{l} \doteq L : Ls \end{array} \right\} \\
\quad \texttt{var kill} = \texttt{function()\{}\mathtt{b_k}\texttt{\};} \\
\quad \left\{ \begin{array}{l} \exists K, K_p. \left(\begin{array}{l} \mathsf{obj}_L(\texttt{kill} : K, @frozen : \texttt{false}) * K \leftarrow s_8 * s_8 \doteq \{(L, \texttt{kill})\} * \\ l_{op} \leftarrow s_9 * s_9 \doteq s_5 \cup \{(K_p, @proto)\} * l_{fp} \leftarrow s_6 \cup \{(K, @proto)\} * \\ L \leftarrow \{(A, @scope), (K, @scope)\} * \\ \mathsf{scopeBpsUpd}(Ls, ss_1, ss_2, \{(K, @scope)\}) * \\ \mathsf{fullobj}_{K_p}(@proto : l_{op}) * K_p \leftarrow \{(K, \texttt{prototype})\} * \\ \mathsf{newfun}_K(\mathbf{l}, \_, b_k, K_p) \end{array} \right) * \mathbf{l} \doteq L : Ls \end{array} \right\} \\
\quad [\text{Frames}] \\
\quad \left\{ \begin{array}{l} \mathsf{obj}_L(\texttt{access} : A, \texttt{kill} : K) * A \leftarrow s_7 * K \leftarrow s_8 * l_{op} \leftarrow s_9 * \mathbf{l} \doteq L : Ls \end{array} \right\} \\
\quad \texttt{return } \{\texttt{access} : \texttt{access}, \texttt{kill} : \texttt{kill}\}; \\
\quad \left\{ \begin{array}{l} \exists O. \left(\begin{array}{l} \mathsf{obj}_L(\texttt{access} : A, \texttt{kill} : K) * \\ \mathsf{fullobj}_O(@proto : l_{op}, \texttt{access} : A, \texttt{kill} : K) * \\ A \leftarrow s_7 \cup \{(O, \texttt{access}\} * K \leftarrow s_8 \cup \{(O, \texttt{kill})\} * \\ l_{op} \leftarrow s_9 \cup \{(O, @proto)\} * O \leftarrow \{\} * \mathbf{r} \doteq O \end{array} \right) * \mathbf{l} \doteq L : Ls \end{array} \right\} \\
\quad [\text{Frames}] \\
\quad \left\{ \begin{array}{l} \mathsf{obj}_R(@body : \lambda\texttt{ref}.\{\ldots\}, @scope : Ls) * \\ \exists L, A, A_p, K, K_p, O. \\ \mathsf{fullobj}_L \left(\begin{array}{l} \texttt{ref} : V, @this : \_, @proto : \texttt{null}, @frozen : \texttt{false}, \\ \texttt{protected} : V, \texttt{access} : A, \texttt{kill} : K \end{array} \right) * \\ \mathsf{fullobj}_{A_p}(@proto : l_{op}) * \mathsf{fullobj}_{K_p}(@proto : l_{op}) * \\ \mathsf{newfun}_A(\mathbf{l}, \texttt{field}, b_a, A_p) * \mathsf{newfun}_K(\mathbf{l}, \_, b_k, K_p) * \\ \mathsf{fullobj}_O(@proto : l_{op}, \texttt{access} : A, \texttt{kill} : K) * \\ l_{op} \leftarrow s_3 \cup \{(A_p, @proto), (K_p, @proto), (O, @proto)\} * \\ l_{fp} \leftarrow s_4 \cup \{(A, @proto), (K, @proto)\} * \\ A \leftarrow \{(L, \texttt{access}), (O, \texttt{access})\} * A_p \leftarrow \{(A, \texttt{prototype}) * \\ K \leftarrow \{(L, \texttt{kill}), (O, \texttt{kill})\} * K_p \leftarrow \{(K, \texttt{prototype}) * \\ V \leftarrow s_1 \cup \{(L, \texttt{ref}), (L, \texttt{protected})\} * \\ \mathsf{scopeBpsUpd}(Ls, ss, ss_2, \{(A, @scope), (K, @scope))\}) * \\ L \leftarrow \{(A, @scope), (K, @scope)\} * O \leftarrow \{\} * \mathbf{r} \doteq O * \mathbf{l} \doteq L : Ls \end{array} \right\} \\
\}
\end{array}
$$

Figure 5.2: Caretaker main body proof

$$
\begin{array}{l}
\texttt{var kill} = \texttt{function}()\{ \\
\quad \left\{ \begin{array}{l} (K, @body) \mapsto \lambda\_.\{b_k\} * (K, @scope) \mapsto L : Ls * \\ \exists L'.\mathbf{l} \doteq L' : L : Ls * L' \leftarrow \{\} * \\ \mathsf{fullobj}_{L'}(@proto : \texttt{null}, @frozen : \texttt{false}, @this : \_) * \\ \mathsf{obj}_{L}(\texttt{protected} : V, @frozen : \texttt{false}) \end{array} \right\} \\
\quad [\text{Frame/exists}] \\
\quad \left\{ \begin{array}{l} obj_{L'}(\texttt{protected} : \varnothing, @proto : \texttt{null}) * \\ obj_{L}(\texttt{protected} : V, @frozen : \texttt{false}) * \\ \mathbf{l} \doteq L' : L : Ls \end{array} \right\} \\
\quad \texttt{protected} = \texttt{null}; \\
\quad \left\{ \begin{array}{l} obj_{L'}(\texttt{protected} : \varnothing, @proto : \texttt{null}) * \\ obj_{L}(\texttt{protected} : \texttt{null}, @frozen : \texttt{false}) * \\ \mathbf{l} \doteq L' : L : Ls * \mathbf{r} \doteq \texttt{null} \end{array} \right\} \\
\quad [\text{Frame/exists}] \\
\quad \left\{ \begin{array}{l} (K, @body) \mapsto \lambda\_.\{b_k\} * (K, @scope) \mapsto L : Ls * \\ \exists L'.\mathbf{l} \doteq L' : L : Ls * L' \leftarrow \{\} * \\ \mathsf{fullobj}_{L'}(@proto : \texttt{null}, @frozen : \texttt{false}, @this : \_) * \\ \mathsf{obj}_{L}(\texttt{protected} : \texttt{null}, @frozen : \texttt{false}) * \mathbf{r} \doteq \texttt{null} \end{array} \right\} \\
\}
\end{array}
$$

Figure 5.3: Caretaker `kill` function body

$$
\begin{array}{l}
\texttt{var access} = \texttt{function}(\texttt{field})\{ \\
\quad \left\{ \begin{array}{l} (A, @body) \mapsto \lambda\texttt{field}.\{b_a\} * (A, @scope) \mapsto L : Ls * \\ \exists L'.\mathbf{l} \doteq L' : L : Ls * L' \leftarrow \{\} * \beta(X, L', \texttt{field}, s) * \\ \mathsf{fullobj}_{L'}(@this : \_, @proto : \texttt{null}, @frozen : \texttt{false}, \texttt{field} : X) * \\ (L, \texttt{protected}) \mapsto P * P \not\doteq \texttt{null} * P \dot{\in} \mathcal{L} * X \dot{\in} \mathcal{X}^{\mathrm{U}} \end{array} \right\} \\
\quad [\text{Frame/exists}] \\
\quad \left\{ \begin{array}{l} \mathsf{obj}_{L'}(\texttt{protected} : \varnothing, @proto : \texttt{null}, \texttt{field} : X) * \\ (L, \texttt{protected}) \mapsto P * P \not\doteq \texttt{null} * P \dot{\in} \mathcal{L} * \\ X \dot{\in} \mathcal{X}^{\mathrm{U}} * \mathbf{l} \doteq L' : L : Ls \end{array} \right\} \\
\quad \texttt{return protected}[\texttt{field}]; \\
\quad \left\{ \begin{array}{l} \mathsf{obj}_{L'}(\texttt{protected} : \varnothing, @proto : \texttt{null}, \texttt{field} : X) * \\ (L, \texttt{protected}) \mapsto P * P \not\doteq \texttt{null} * P \dot{\in} \mathcal{L} * \\ X \dot{\in} \mathcal{X}^{\mathrm{U}} * \mathbf{l} \doteq L' : L : Ls * \mathbf{r} \doteq P{\cdot}X \end{array} \right\} \\
\quad [\text{Frame/exists}] \\
\quad \left\{ \begin{array}{l} (A, @body) \mapsto \lambda\texttt{field}.\{b_a\} * (A, @scope) \mapsto L : Ls * \\ \exists L'.\mathbf{l} \doteq L' : L : Ls * L' \leftarrow \{\} * \beta(X, L', \texttt{field}, s) * \\ \mathsf{fullobj}_{L'}(@this : \_, @proto : \texttt{null}, @frozen : \texttt{false}, \texttt{field} : X) * \\ (L, \texttt{protected}) \mapsto P * P \not\doteq \texttt{null} * P \dot{\in} \mathcal{L} * X \dot{\in} \mathcal{X}^{\mathrm{U}} * \mathbf{r} \doteq P{\cdot}X \end{array} \right\} \\
\}
\end{array}
$$

Figure 5.4: Caretaker `access` function body

5.2 Membrane

The Membrane pattern is an enhanced version of Caretaker, but in addition to protecting all interactions with a particular object, a Membrane also protects all subsequent products of interactions with the protected object.

This is achieved by wrapping all returned objects in a Membrane Reference that shares its 'kill switch' with the parent Membrane. When the Membrane is killed, all references to the parent object and any references that have been passed as a result of the parent are revoked simultaneously.

A simple Membrane implementation is shown in Figure 5.5.

```
var Membrane = function(ref) {
  var killed = false;

  var MembraneRef = function(ref) {
    var access = function(field) {
      if(!killed) return MembraneRef(ref[field]);
    }
    if(primitive(ref)) { return ref; } else { return access; }
  }

  var access = MembraneRef(ref);
  var kill = function() { killed = true; }
  return {access: access, kill: kill};
}
```

Figure 5.5: Implementation of the Membrane pattern

In order to prove the correctness of the Membrane, we take a more principled approach than we did for the considerably simpler Caretaker. We begin by defining predicates that will help us simplify the desired specifications for the Membrane functionality.

We begin by defining the assertion for an instance of a Membrane, along with the more refined definitions of an alive and a dead membrane. These definitions encapsulate the Membrane's activation record, M, on which the kill switch and supporting functions are initially defined, one of which is the MembraneRef constructor, F_{MR}, which is used recursively by the Membrane implementation.

$$\begin{aligned} \mathit{MembraneInstance}(M, F_{MR}, F_K) \triangleq\ & \mathsf{obj}_M(\texttt{ref} : \_, @\mathit{this} : \_, @\mathit{proto} : \texttt{null}, \texttt{MembraneRef} : F_{MR}, \texttt{kill} : F_K) * \\ & \mathsf{newfun}_{F_{MR}}(M : \_, \texttt{ref}, \lambda_{MR}, \_) \\ \mathit{alive}(M, F_{MR}, F_K) \triangleq\ & \mathit{MembraneInstance}(M, F_{MR}, F_K) * (M, \mathit{killed}) \mapsto \texttt{false} \\ \mathit{dead}(M, F_{MR}, F_K) \triangleq\ & \mathit{MembraneInstance}(M, F_{MR}, F_K) * (M, \mathit{killed}) \mapsto \texttt{true} \end{aligned}$$

The kill function F_K for the Membrane M, is simply defined as:

$$\mathit{kill}(F_K, M) \triangleq \mathsf{fun}_{F_K}(M : \_, \_, \lambda_K, \_)$$

We next define predicates for the MembraneRef objects, MR, that are created as a result of using the Membrane, M. Each MembraneRef protects a potentially different object, r. F_A is the corresponding access function for the MembraneRef. Finally, it is specified that the only objects that may point at r are in the sets S, references in the trusted code, and T, references created by the Membrane.

Note that the newfun predicate cannot be used to describe F_A here, since it is passed into untrusted code. Although the untrusted code may add new fields to the function, it will have no impact upon

the proof, as the function only accesses variables through its scope chain. In the case of additional fields being added, the newobj part of the newfun predicate would prevent proof of any such code.

$$MRef_{MR}(M, r, F_A, S, T) \triangleq \mathsf{fun}_{F_A}(MR : M : \_, \texttt{field}, \lambda_A, \_) * \\ \mathsf{fullobj}_{MR}(\texttt{ref} : r, \texttt{access} : F_A, @this : \_, @proto : \texttt{null}) * r \leftarrow S \cup T$$

Finally, we must define the set of references, T, created by the Membrane which are permitted to have references to protected objects. The Ls parameter is a list of $MRef$ objects constructed by the Membrane, M. The list Ls is updated in the specification of `MembraneRef`.

$$MRset_M([], \{(M, \texttt{ref})\}) \triangleq \oslash \\ MRset_M(L : Ls, Ts') \triangleq MRset_M(Ls, Ts) * Ts' \doteq Ts \cup \{(L, \texttt{ref})\}$$

We now specify the properties required of Membrane and its associated functions. The `Membrane` function should take a reference, and produce an alive Membrane, with a kill function, an access function for the reference, and return these in a simple object.

$$\left\{ \ (\_, \texttt{ref}) \mapsto ref * ref \leftarrow S * ref \dot{\in} \mathcal{L} * \mathbf{l} \doteq Ls \ \right\} \\ \texttt{Membrane(ref)}; \\ \left\{ \begin{array}{l} (\_, ref) \mapsto ref * ref \dot{\in} \mathcal{L} * \mathbf{l} \doteq Ls * \\ \exists M, F_K, F_A, F_{MR}, MR, L. \left(\begin{array}{l} alive(M, F_{MR}, F_K) * kill(F_K, M) * MRef_{MR}(M, ref, F_A, S, T) * \\ MRset([MR], T) * \mathsf{fullobj}_L(\texttt{access} : F_A, \texttt{kill} : F_K) * \mathbf{r} \doteq L \end{array} \right) \end{array} \right\}$$

The kill function specification is self-explanatory.

$$\left\{ \ (\_, \texttt{kill}) \mapsto F_K * kill(F_K, M) * (dead(M, F_{MR}, F_K) \vee alive(M, F_{MR}, F_K)) \ \right\} \\ \texttt{kill}(); \\ \left\{ \ (\_, \texttt{kill}) \mapsto F_K * kill(F_K, M) * dead(M, F_{MR}, F_K) \ \right\}$$

The internal MembraneRef function takes a Membrane, the set of objects constructed by the membrane, and a reference to be protected. It leaves the Membrane unchanged, constructs a new MRef object, extends the list of MRef objects constructed, and returns the MRef's associated access function.

$$\left\{ \begin{array}{l} (\_, \texttt{MembraneRef}) \mapsto F_{MR} * alive(M, F_{MR}, F_K) * MRset_M(Ls, T) * \\ (\_, \texttt{ref}) \mapsto ref * ref \dot{\in} \mathcal{L} * ref \leftarrow S \cup T \end{array} \right\} \\ \texttt{MembraneRef(ref)}; \\ \left\{ \begin{array}{l} (\_, \texttt{MembraneRef}) \mapsto F_{MR} * alive(M, F_{MR}, F_K) * \\ \exists R, F_A. \left(\ MRset_M(R : Ls, T') * MRef_R(M, ref, F_A, S, T') * \mathbf{r} \doteq F_A \ \right) * \\ (\_, \texttt{ref}) \mapsto ref * ref \dot{\in} \mathcal{L} \end{array} \right\}$$

The access function is inherently associated with an MRef object, which it must have permission to. The field being accessed must be a user-accessible field on the object protected by the MRef and it must point to an object.

$$\left\{ \begin{array}{l} (\_, \texttt{access}) \mapsto F_{A_1} * alive(M, F_{MR}, F_K) * MRef_{R_1}(M, ref, F_{A_1}, S_1, T_1) * MRset(Ls, T_1) * \\ (\_, \texttt{field}) \mapsto F * F \dot{\in} \mathcal{X}^{\mathrm{U}} * \gamma(\_, ref{\cdot}F, V) * V \dot{\in} \mathcal{L} * V \leftarrow S_2 \end{array} \right\} \\ \texttt{access(field)}; \\ \left\{ \begin{array}{l} (\_, access) \mapsto F_{A_1} * alive(M, F_{MR}, F_K) * MRef_{R_1}(M, ref, F_{A_1}, S_1, T_2) * \\ \exists R_2, F_{A_2}. \left(\ MRset(R_2 : Ls, T_2) * MRef_{R_2}(M, V, F_{A_2}, S_2, T_2) * \mathbf{r} \doteq F_{A_2} \ \right) * \\ (\_, \texttt{field}) \mapsto F * F \dot{\in} \mathcal{X}^{\mathrm{U}} * \gamma(\_, ref{\cdot}F, V) * V \dot{\in} \mathcal{L} \end{array} \right\}$$

Proof outlines for these specifications are shown in figures 5.6 and 5.7.

$$
\begin{array}{l}
\texttt{var Membrane = function(ref) \{} \\
\quad \left\{ \begin{array}{l} \mathsf{fullobj}_M \left(\begin{array}{l} \texttt{ref} : ref, @this : \_, @proto : \texttt{null}, \texttt{killed} : \texttt{undefined}, \\ \texttt{MembraneRef} : \texttt{undefined}, \texttt{access} : \texttt{undefined}, \texttt{kill} : \texttt{undefined} \end{array} \right) * \\ MRset_M([], T) * ref \leftarrow S \cup T * ref \dot{\in} \mathcal{L} * \mathbf{l} \doteq M : Ls \end{array} \right\} \\
\quad \texttt{var killed = false;} \\
\quad \texttt{var MembraneRef = function(ref) \{...\}} \\
\quad \texttt{var kill = function() \{...\}} \\
\quad \left\{ \begin{array}{l} alive(M, F_{MR}, F_K) * \mathsf{newobj}_M(\texttt{ref}, @this, @proto, \texttt{killed}, \texttt{MembraneRef}, \texttt{access}, \texttt{kill}) * \\ kill(F_K, M) * (M, \texttt{access}) \mapsto \texttt{undefined} * \\ MRset_M([], T) * ref \leftarrow S \cup T * ref \dot{\in} \mathcal{L} * \mathbf{l} \doteq M : Ls \end{array} \right\} \\
\quad \texttt{var access = MembraneRef(ref);} \\
\quad \left\{ \begin{array}{l} alive(M, F_{MR}, F_K) * \mathsf{newobj}_M(\texttt{ref}, @this, @proto, \texttt{killed}, \texttt{MembraneRef}, \texttt{access}, \texttt{kill}) * \\ kill(F_K, M) * (M, \texttt{access}) \mapsto F_A * \\ MRset_M([MR], T') * MRef_{MR}(M, ref, F_A, S, T') * ref \dot{\in} \mathcal{L} * \mathbf{l} \doteq M : Ls \end{array} \right\} \\
\quad \texttt{return \{access: access, kill: kill\};} \\
\quad \left\{ \begin{array}{l} alive(M, F_{MR}, F_K) * \mathsf{newobj}_M(\texttt{ref}, @this, @proto, \texttt{killed}, \texttt{MembraneRef}, \texttt{access}, \texttt{kill}) * \\ kill(F_K, M) * (M, \texttt{access}) \mapsto F_A * \\ MRset_M([MR], T') * MRef_{MR}(M, ref, F_A, S, T') * ref \dot{\in} \mathcal{L} * \\ \mathsf{fullobj}_L(\texttt{access} : F_A, \texttt{kill} : F_K) * \mathbf{l} \doteq M : Ls * \mathbf{r} \doteq L \end{array} \right\} \\
\texttt{\}}
\end{array}
$$

$$
\begin{array}{l}
\texttt{var kill = function() \{} \\
\quad \{ \ \exists A. \mathbf{l} \doteq A : M : \_ * \mathsf{fullobj}_A(@proto : \texttt{null}, @this : \_) * (alive(M, F_{MR}, F_K) \vee dead(M, F_{MR}, F_K)) \ \} \\
\quad \texttt{killed = true;} \\
\quad \{ \ \exists A. \mathbf{l} \doteq A : M : \_ * \mathsf{fullobj}_A(@proto : \texttt{null}, @this : \_) * dead(M, F_{MR}, F_K) \ \} \\
\texttt{\}}
\end{array}
$$

Figure 5.6: Proof outlines for `Membrane` and its nested `kill` function

```
var MembraneRef = function(ref) {
```
$$\left\{ \exists R. \left(\begin{array}{l} alive(M, F_{MR}, F_K) * \mathsf{fullobj}_R(\texttt{ref} : ref, @this : \_, @proto : \texttt{null}, \texttt{access} : \texttt{undefined}) * \\ ref \dot{\in} \mathcal{L} * MRset_M(R : Ls, T') * ref \leftarrow S \cup T' * \mathbf{l} \doteq R : M : Ls \end{array} \right) \right\}$$
```
    var access = function(field) {...}
```
$$\left\{ \exists R, F_A. \left(\begin{array}{l} alive(M, F_{MR}, F_K) * \mathsf{fullobj}_{MR}(\texttt{ref} : ref, @this : \_, @proto : \texttt{null}, \texttt{access} : F_A) * \\ \mathsf{newfun}_{F_A}(R : M : Ls, \texttt{field}, \lambda_A, \_) * ref \dot{\in} \mathcal{L} * \\ MRset_M(R : Ls, T') * ref \leftarrow S \cup T' * \mathbf{l} \doteq R : M : Ls \end{array} \right) \right\}$$
```
    if(primitive(ref)) {...} else {
        return access;
    }
```
$$\left\{ \exists R, F_A. \left(\begin{array}{l} alive(M, F_{MR}, F_K) * \mathsf{fullobj}_R(\texttt{ref} : ref, @this : \_, @proto : \texttt{null}, \texttt{access} : F_A) * \\ \mathsf{newfun}_{F_A}(R : M : Ls, \texttt{field}, \lambda_A, \_) * ref \dot{\in} \mathcal{L} * \\ MRset_M(R : Ls, T') * ref \leftarrow S \cup T' * \mathbf{l} \doteq R : M : Ls * \\ \mathbf{r} \doteq F_A \end{array} \right) \right\}$$
[subst]
$$\left\{ \exists R, F_A. \left(\begin{array}{l} alive(M, F_{MR}, F_K) * MRset_M(R : Ls, T') * MRef_R(M, ref, F_A, S, T') * \\ ref \dot{\in} \mathcal{L} * rv \doteq F_A * \mathbf{l} \doteq R : M : Ls \end{array} \right) \right\}$$
```
}
```

```
var access = function(field) {
```
$$\left\{ \begin{array}{l} \exists A. \mathbf{l} \doteq A : R_1 : M : \_ * \mathsf{fullobj}_A(\texttt{field} : field, @proto : \texttt{null}, @this : \_) * \\ alive(M, F_{MR}, F_K) * MRef_{R_1}(M, ref, F_{A_1}, S_1, T_1) * MRset(Ls, T_1) * \\ field \dot{\in} \mathcal{X}^{\mathrm{U}} * (ref, field) \mapsto V * V \dot{\in} \mathcal{L} * V \leftarrow S_2 \end{array} \right\}$$
```
    if(!killed) {
```
$$\left\{ \begin{array}{l} \exists A. \mathbf{l} \doteq A : R_1 : M : \_ * \\ alive(M, F_{MR}, F_K) * MRef_{R_1}(M, ref, F_{A_1}, S_1, T_1) * MRset_M(Ls, T_1) * \\ (A, \texttt{field}) \mapsto field * field \dot{\in} \mathcal{X}^{\mathrm{U}} * \\ (ref, field) \mapsto V * V \dot{\in} \mathcal{L} * V \leftarrow S_2 \cup T_1 \end{array} \right\}$$
```
        return MembraneRef(ref[field]);
```
$$\left\{ \exists A, R_2, F_{A_2}. \left(\begin{array}{l} \mathbf{l} \doteq A : R_1 : M : \_ * alive(M, F_{MR}, F_K) * \\ MRef_{R_1}(M, ref, F_{A_1}, S_1, T_1) * MRef_{R_2}(M, V, F_{A_2}, S_2, T_2) * \\ MRset_M(R_2 : Ls, T_2) * \mathbf{r} \doteq F_{A_2} * \\ (A, \texttt{field}) \mapsto field * field \dot{\in} \mathcal{X}^{\mathrm{U}} * (ref, field) \mapsto V * V \dot{\in} \mathcal{L} \end{array} \right) \right\}$$
```
    }
```
$$\left\{ \exists A, R_2, F_{A_2}. \left(\begin{array}{l} \mathbf{l} \doteq A : R_1 : M : \_ * \mathsf{fullobj}_A(\texttt{field} : field, @proto : \texttt{null}, @this : \_) * \\ alive(M, F_{MR}, F_K) * MRef_{R_1}(M, ref, F_{A_1}, S_1, T_2) * \\ MRset(R_2 : Ls, T_2) * MRef_{R_2}(M, V, F_{A_2}, S_2, T_2) * \mathbf{r} \doteq F_{A_2} * \\ field \dot{\in} \mathcal{X}^{\mathrm{U}} * (ref, field) \mapsto V * V \dot{\in} \mathcal{L} \end{array} \right) \right\}$$
```
}
```

Figure 5.7: Proof outlines for the nested `MembraneRef` and `access` functions

5.3 Example use of Caretaker and Membrane

Assume we want to allow an untrusted program read access to `cookies` until potentially sensitive data is stored in them by `updateCookies`.

```
cookies.password = '';
var cookieMembrane = Membrane(cookies);
var updateCookies = function(name, value) {
  cookieMembrane.kill();
  cookies[name] = value;
}

reval(maliciousCode, {cookies: cookieMembrane.access});
if (userHasEnteredPassword) {
  updateCookies('password', userPassword);
}
```

The malicious code is able to access the cookies through the `cookieMembrane.access` function[1] passed into it as the `cookies` parameter of the *imports* object. Note that `maliciousCode` is unable to access any of the global variables such as `userPassword`.

This specification for `maliciousCode` is *maximally malicious*. The *chaos* predicate traverses every path that can be followed from l, and does not specify what changes have been made.

$$\begin{array}{l}\{\ \mathbf{1} \doteq [l] * traverse(l)\ \} \\ \texttt{maliciousCode}(); \\ \{\ \mathbf{1} \doteq [l] * chaos(l)\ \}\end{array}$$

Despite this malicious specification of `maliciousCode`, we can prove that the global password field remains untouched.

To do this, we specify a general verification condition that privileged objects are not passed to the untrusted code, by ensuring that the imports object is not reachable by traversing $\leftarrow$ assertions from the privileged objects.

In cases where untrusted code is allowed restricted access to privileged objects, there will be an object which is only pointed to by @scope fields of a (some) function(s). In this case, the privileged object is protected according to the specifications of those function(s).

For example, a backpointer path for the `cookies` object is:

$$\begin{aligned} cookies &\leftarrow \{(l_g, \texttt{cookies}), (membraneref, \texttt{ref})\} * \\ membrane &\leftarrow \{(access, @scope)\} * \\ access &\leftarrow \{(imports, \texttt{cookies})\} \end{aligned}$$

From the middle line we can see that the cookies are protected by the membrane's access function.

[1] Note that one disadvantage with the Membrane is that the field accessor now has function-call syntax. The upcoming ES6 standard has a *Proxies* feature that allows for intercepting standard syntax on an object and redirecting the input to a custom event handler. This will help make Membrane access functions *transparent* to the user

Chapter 6

Evaluation

The nature of this project means that it is somewhat hard to evaluate the efficacy of the developed logic. Nonetheless, we can judge aspects based on: direct comparison with other published works; discussion of the applicability of the extension to separation logic to other languages;

6.1 Operational Semantics

The SES Operational Semantics were developed from a mixture of sources of specification material, from the original JavaScript Operational Semantics, from a *very* loosely described informational document about SES, through to examination of the source code of the JavaScript implementation of SES. Where these sources differed, we opted to match the behaviour of the JavaScript implementation.

Since completion of the semantics, a technical report accompanying [28] was discovered to contain a well-defined version of the SES Operational Semantics (hereon Taly's semantics).

Direct comparison of our Operational Semantics and Taly's showed general equivalence between the two languages albeit Taly's made fewer generalisations and axiomatized more of the built-in commands.

Notably, the command whose precise semantics we were least sure of, the restricted evaluation command, was found to be equivalent to those published by Taly.

6.2 Program Logic

As discussed in section 4.1.4, the $\leftarrow$ operator cannot be a pure operator as a result of the unintentional intersection of backpointer sets that would occur as demonstrated in figure 4.1. This issue was remedied by giving the $\leftarrow$ a footprint in the heap.

Depending on viewpoint, this solution may be considered a nice way around an otherwise serious issue with the logic, or a horrendous *hack* that depends upon an internal variable defined upon the abstract heap just for the logic.

Ideally, more effort should have been spent to find a more general way to introduce a footprint to this operator.

6.3 Program Proofs

The two program proofs presented in chapter 5 demonstrate that depending upon the purpose and generation method, proof derivations can be either hugely verbose and hard to follow if produced mechanically with little abstraction, or if some simple abstractions are applied, can be followed relatively easily with a casual reading.

Dr. Mark Miller from Google, the designer of the SES language, visited Imperial to give a seminar on the future of SES on the web in March 2013. We met to discuss verification of SES programs using this logic. We walked through the proof of the Caretaker pattern to ensure that the model of the language and the verification logic were consistent with his own understanding of the language.

The meeting was beneficial as it confirmed that the model of the language and the logic were indeed in line with his understanding of JavaScript and SES. The meeting was also beneficial for revealing that the definition of an early version of scopeBps was highly unintuitive, although the aim was correct.

6.4 'Groundwork'

It may have been noted by the reader that although we have defined `freeze`, `def`, `reval`, and $-\!\!\circledast$, they are not actually extensively used in this project.

`freeze` and `def` were defined for use for passing immutable interfaces to untrusted code. Their definition and axiomatization was driven by an aim to prove the makeContractHost algorithm[2]. Unfortunately, this aim was dropped when it transpired that earlier, *easier* program proofs would take far longer than expected to produce. It is hoped that a proof could be developed before the presentation of this project.

`reval` is mentioned in passing during the discussion of the Caretaker and Membrane proofs, using a combination of them and `reval` guarantees the isolation properties proven on these design patterns. Untrusted code that is executed outside of a `reval` command can trivially reach the protected object by traversing the heap structure from the global object.

The $-\!\!\circledast$ logical operator was produced with the knowledge that the conventional $-\!\!*$ operator is used to generate a weakest precondition in the proof of soundness of the Assignment inference rule.

Chapter 7

Conclusions & Future Work

7.1 Conclusions

- We have specified the semantics of a model of the SES language.
- We have produced the first program logic for SES
- We have produced the first separation logic-based program logic that can reason about object capability security through backpointers
- We have produced the first proofs for some object capability programs
- We have shown that proofs in the program logic match closely match the intuitions of a programmer who has extensively studied the language and program
- And, along the way, we have discovered the duality of uses for the wand operator

7.2 Future Work

There are many directions in which work on the SES language and verification logic can be pursued to follow the developments of the JavaScript language, the distributed variant of SES, or to improve and understand the logic further.

7.2.1 Extend the SES Language Model

This model of the SES language was based upon old and generalised ES3 semantics, in order to demonstrate the extensions to the logic in a simple way. However, a project is underway within the Reliable Web research group to precisely specify the ES5.1 specification. Properly formalising this model of SES on top of these new semantics using the same theorem proving tools would give a boost to the accuracy of the logic and provide benefits from the tooling that is being built around these semantics and logic.

The SES language is only one portion of a bigger plan for web programming, with the emergence of JavaScript on the server, there is a demand for communications primitives that are easier to use. One extreme of this is the extension of the object model of the language across the network to other machines. This is possible through a syntax-transparent communication library. In this case there is a definite motivation for the SES component of DrSES.

This new, distributed language poses new issues in program verification since communication also has to be considered. A potential method for reasoning about programs written in this style may involve a fusion of the SES logic with a system suitable for reasoning about communications such as Session Types.

And finally, there's always a large number of other interesting algorithms to verify, Sealable References are another common sample program often used in research papers into object capability-style languages. As a challenge during his visit, he posed us to prove all the algorithms he uses in his presentations – several of these would prove quite a challenge as they rely on complex cascades of callback functions.

7.2.2 Logic

After the project had begun, we noted that there was newly published material [19] that also introduces non-local heap state to the logical assertion satisfaction relations. However, the paper does not warrant the use of $-\!\!*$, so they left it undefined in their logic.

We believe there is further research to be made into the definitions of $-\!\!*$ and $-\!\!\boxtimes$ in the presence of non-local heap state. We consider that it may be possible to define a more general version of the wand to support all of its common use cases as well as maintaining the right-adjoint property.

We have not considered how the new operators act in the presence of negation, it is already considered that negation acts in unintuitive ways in separation logic. It may be interesting to further see how the backpointer reacts under negation.

If we can extend the logic to support some aspects of non-locality, can we add another tier to the heap states to reduce the coarseness of the local vs. global reasoning. For example, is it possible to split the heap into three state environments of local, trusted, untrusted? What further operators would we require to express useful assertions in this logic?

7.2.3 Verification of the JavaScript-based SES implementation

The implementation of SES as a layer on top of JavaScript also poses its own interesting problems. SES inherently depends upon the JavaScript specification as its basis, but the JavaScript specification process is also directed to maintaining the properties required to enable SES to be implemented securely. These properties are usually very subtle, such as ensuring that the language-defined data structures and primitives do not inadvertently allow for pointer leakage or covert channels of communication within the language.

As an indirect result of this project a potential bug [3] has been discovered in the upcoming ECMAScript specification by the ECMAScript research group at Imperial. This demonstrates that although the specification committee should now have a good insight into the security implications of the language's design decisions, errors are still being made as a result of a poor, informal description of the security properties that ought to hold for the language. With a formal specification of these properties, we could prove that the formally specified language definitely does hold these properties.

A proof that the SES implementation is correct is currently well beyond the capabilities of the JavaScript program logic, as the SES implementation reasons about the particular implementation-specific semantics of the JavaScript engine on which it is run, and attempts to fix any semantics that it deems unsafe.

Program logics to date assume that there is one correct semantics for a language under which a program runs, JavaScript poses the unique possibility where a program may run under similar, but subtly different semantics. Research into extension of the existing JavaScript program logic to support variable semantics would be a novel research project.

7.3 Automated Program Verification

In general, the problem of finding specifications for programs is undecidable, however because separation logic is structural, this provides additional power for automated reasoning programs to leverage. Theories behind Symbolic Execution [6] and Biabduction [8] for separation logic were developed by O'Hearn et al. as a means of analysing the shapes of footprints of assertions.

Applications of these methods have shown them to be suitable for use in practical and scalable automated reasoning tools for many languages including C and Java [4, 5, 29, 10].

It is anticipated that such theory can be applied to the JavaScript separation logic to provide realistic tools for automated JavaScript verification.

7.3.1 Comparisons with Other Techniques

Considerable amounts of research has been performed into dynamic information flow analysis for JavaScript programs through instrumentation of the interpreter [18]. Abstract interpretation for information flow analysis on JavaScript has also been attempted [9] successfully. It was noted during the construction of figure 5.2 that the building of the backpointer sets 'felt' very much like the execution of abstract interpretation algorithms as discussed (and practiced) in the 4th Year Course [14, 24]. Drawing a more formal comparison between the two techniques may assist in the development or adoption of already-existing techniques from the Abstract Interpretation field into this.

It has been suggested [11] that Ownership Types would be a suitable means of verifying the security properties of SES. It would be interesting to see if Ownership Types for this purpose can be expressed using the logic developed in the course of this project.

Bibliography

[1] Making javascript safe from advertising. URL http://www.adsafe.org/. Accessed: 18-6-2013.

[2] makecontracthost.js - ses contract host algorithm. URL http://code.google.com/p/es-lab/source/browse/trunk/src/ses/contract/makeContractHost.js.

[3] Ecmascript bug 1444 - 8.3.12 incomplete in the presence of prototype changes and property deletions. URL https://bugs.ecmascript.org/show_bug.cgi?id=1444. Accessed: 18-06-2013.

[4] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.

[5] J. Berdine, B. Cook, and S. Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV*, 2011.

[6] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, APLAS'05, pages 52–68, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-29735-9, 978-3-540-29735-2. doi: 10.1007/11575467_5. URL http://dx.doi.org/10.1007/11575467_5.

[7] R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine intelligence*, 7:23–50, 1972.

[8] C. Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL '09: Proc. of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 289–300, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: http://doi.acm.org/10.1145/1480881.1480917.

[9] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 50–62, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542483. URL http://doi.acm.org/10.1145/1542476.1542483.

[10] D. Distefano and M. Parkinson. jStar: towards practical verification for Java. In *OOPSLA '08*, pages 213–226. ACM, 2008.

[11] Sophia Drossopoulou, David Clarke, and James Noble. Roles for Owners - Work in progress -. In *IWACO 2011*, ACM Digital Library, July 2011. URL http://pubs.doc.ic.ac.uk/rolesForOwners/.

[12] R.W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19:19–32, 1967.

[13] P. Gardner, S. Maffeis, and G. Smith. Towards a program logic for JavaScript. In *POPL*, 2012.

[14] C. Hankin and H. Wiklicky. C480 – program analysis. Imperial College London Lecture Notes, 2013.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259.

[16] C. A. R. Hoare. Proof of a program: Find. *Commun. ACM*, 14(1):39–45, January 1971. ISSN 0001-0782. doi: 10.1145/362452.362489. URL http://doi.acm.org/10.1145/362452.362489.

[17] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 14–26, New York, NY, USA, 2001. ACM. ISBN 1-58113-336-7. doi: 10.1145/360204.375719. URL http://doi.acm.org/10.1145/360204.375719.

[18] Seth Just, Alan Cleary, Brandon Shirley, and Christian Hammer. Information flow analysis for javascript. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, PLASTIC '11, pages 9–18, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1171-7. doi: 10.1145/2093328.2093331. URL http://doi.acm.org/10.1145/2093328.2093331.

[19] Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 561–574, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429134. URL http://doi.acm.org/10.1145/2429069.2429134.

[20] S. Maffeis and A. Taly. Language-based isolation of untrusted javascript. In *Computer Security Foundations Symposium, 2009. CSF '09. 22nd IEEE*, pages 77 –91, july 2009. doi: 10.1109/CSF.2009.11.

[21] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In G. Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 307–325. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-89329-5. doi: 10.1007/978-3-540-89330-1_22. URL http://dx.doi.org/10.1007/978-3-540-89330-1_22.

[22] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 125 –140, may 2010. doi: 10.1109/SP.2010.16.

[23] M.S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Safe active content in sanitized javascript. Technical report, 2008.

[24] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540654100.

[25] P. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.

[26] J.C. Reynolds. Intuitionistic reasoning about shared mutable data structure. *Millennial perspectives in computer science*, pages 303–321, 2000.

[27] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55 – 74, 2002. doi: 10.1109/LICS.2002.1029817.

[28] A. Taly, Erlingsson, J.C. Mitchell, M.S. Miller, and J. Nagra. Automated analysis of security-critical javascript apis. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 363 –378, may 2011. doi: 10.1109/SP.2011.39.

[29] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.

Appendix A

Notation

Notation: Sorts and Constants.

$H \in \mathcal{L} \times \mathcal{X} \rightharpoonup \mathcal{V}$	Heaps.
$l \in \mathcal{L}_{\texttt{null}} \triangleq \mathcal{L} \cup \{\texttt{null}\}$	Locations.
$L \in \mathcal{S} \triangleq \mathcal{L}^n$	Scope chains.
l_g	Global object.
l_{op}	`Object.prototype`.
$\left\{ \begin{array}{l} @proto, @this, \\ @scope, @body, @bp \end{array} \right\} \in \mathcal{X}^{\mathrm{I}}$	Internal variables.
$x \in \mathcal{X} \triangleq \mathcal{X}^{\mathrm{I}} \uplus \mathcal{X}^{\mathrm{U}}$	Variables.
$\mathtt{x} \in \mathcal{X}^{\mathrm{U}} \subseteq \mathcal{M}$	User variables.
$r \in \mathcal{V}^{\mathrm{R}} \triangleq \mathcal{V} \cup \mathcal{R}$	Return values.
$v \in \mathcal{V} \triangleq \mathcal{V}^{\mathrm{U}} \cup \mathcal{L}_{\texttt{null}} \cup \mathcal{S} \cup \mathcal{F}$	Semantic values.
$\mathtt{v} \in \mathcal{V}^{\mathrm{U}} \triangleq \mathcal{N} \cup \mathcal{M} \cup \mathcal{U} \cup \{\texttt{null}\}$	User values.
$\mathtt{n} \in \mathcal{N}$	Numbers.
$\mathtt{m} \in \mathcal{M}$	Strings.
$\texttt{undefined} \in \mathcal{U}$	Undefined.
$l{\cdot}\mathtt{x} \in \mathcal{R}$	References.
$\lambda\mathtt{x.e} \in \mathcal{F}$	Function code.
$\mathtt{e} \in \mathcal{E}$	Expressions.

Appendix B

Operational Semantics

Auxiliary Functions

$$
\begin{array}{lcll}
\mathsf{obj}(l, l') & \triangleq & l \mapsto \{@proto : l', @frozen : \texttt{false}\} & \\
\mathsf{True}(v) & \triangleq & v \notin \{0, \text{“ ”}, \texttt{null}, \texttt{undefined}, \texttt{false}\} & \\
\mathsf{False}(v) & \triangleq & v \in \{0, \text{“ ”}, \texttt{null}, \texttt{undefined}, \texttt{false}\} & \\
\mathsf{fun}(l', L, \mathtt{x}, \mathtt{e}, l) & \triangleq & l' \mapsto \{@proto : l_{fp}, \texttt{prototype} : l, @scope : L, @body : \lambda \mathtt{x}.\mathtt{e}, @frozen : \texttt{false}\} & \\
\mathsf{This}(H, l{\cdot}x) & \triangleq & l & \text{if } (l, @this) \notin \mathsf{dom}(H) \\
\mathsf{This}(H, r) & \triangleq & \texttt{undefined} & \text{otherwise} \\
\mathsf{SelectProto}(l) & \triangleq & l \quad l \in \mathcal{L} & \\
\mathsf{SelectProto}(v) & \triangleq & l_{op} \quad v \notin \mathcal{L} & \\
\mathsf{getBase}(l, l') & \triangleq & l' \quad l \in \mathcal{L} & \\
\mathsf{getBase}(l, v) & \triangleq & v \quad v \notin \mathcal{L} & \\
\mathsf{act}(l, \mathtt{x}, v, \mathtt{e}, l'') & \triangleq & l \mapsto \{\mathtt{x} : v, @this : l'', @proto : \texttt{null}\} \uplus \mathsf{defs}(\mathtt{x}, l, \mathtt{e}) & \\
\mathsf{auxDef}(H, l, s) & \triangleq & \begin{cases} \texttt{emp} & l \in s \vee l \notin \mathcal{L} \\ (l, @frozen) \mapsto \texttt{true} \cup \bigcup_{(l, x_n) \in H, x_n \in \mathcal{X}^v} \mathsf{auxDef}(H, H(l, x_n), s \cup \{l\}) & \text{otherwise} \end{cases} & \\
\mathsf{RW}(H, l) & \triangleq & H(l, @frozen) = \texttt{false} &
\end{array}
$$

Local Variable Declarations

$$
\begin{array}{l}
\mathsf{defs}(\mathtt{x}, l, \texttt{var y}) \triangleq (l, y) \mapsto \texttt{undefined} \quad \text{if } \mathtt{x} \neq \mathtt{y} \\
\mathsf{defs}(\mathtt{x}, l, \mathtt{e}_1 = \mathtt{e}_2) \triangleq \mathsf{defs}(\mathtt{x}, l, \mathtt{e}_1) \\
\mathsf{defs}(\mathtt{x}, l, \mathtt{e}_1; \mathtt{e}_2) \triangleq \mathsf{defs}(\mathtt{x}, l, \mathtt{e}_1) \cup \mathsf{defs}(\mathtt{x}, l, \mathtt{e}_2) \\
\mathsf{defs}(\mathtt{x}, l, \texttt{if}(\mathtt{e}_1)\{\mathtt{e}_2\}\{\mathtt{e}_3\}) \triangleq \mathsf{defs}(\mathtt{x}, l, \mathtt{e}_2) \cup \mathsf{defs}(\mathtt{x}, l, \mathtt{e}_3) \\
\mathsf{defs}(\mathtt{x}, l, \texttt{while}(\mathtt{e}_1)\{\mathtt{e}_2\}) \triangleq \mathsf{defs}(\mathtt{x}, l, \mathtt{e}_2) \\
\mathsf{defs}(\mathtt{x}, l, \mathtt{e}) \triangleq \texttt{emp} \quad \text{otherwise}
\end{array}
$$

Heap Update $H[H']$

$$
\begin{array}{l}
H[\texttt{emp}] \triangleq H \\
H[(l, x) \mapsto v] \triangleq H \uplus (l, x) \mapsto v \quad \text{if } (l, x) \notin \mathsf{dom}(H) \\
H[(l, x) \mapsto v \uplus H'] \triangleq H[(l, x) \mapsto v][H']
\end{array}
$$

Scope resolution: $\sigma(H, l, x)$**.**

$$
\sigma(H, [\,], x) \triangleq \texttt{null}
$$

$$
\frac{\pi(H, l, x) \neq \texttt{null}}{\sigma(H, l{:}L, x) \triangleq l} \qquad \frac{\pi(H, l, x) = \texttt{null}}{\sigma(H, l{:}L, x) \triangleq \sigma(H, L, x)}
$$

Prototype resolution: $\pi(H, l, x)$**.**

$$
\pi(H, \texttt{null}, x) \triangleq \texttt{null}
$$

$$\frac{(l,x) \in \mathsf{dom}(H)}{\pi(H,l,x) \triangleq l} \qquad \frac{(l,x) \notin \mathsf{dom}(H) \qquad H(l,@proto) = l'}{\pi(H,l,x) \triangleq \pi(H,l',x)}$$

Dereferencing values: $\gamma(H,r)$.

$$\frac{r \neq l{\cdot}\mathtt{x}}{\gamma(H,r) \triangleq r} \qquad \frac{\pi(H,l,\mathtt{x}) = \mathtt{null} \quad l \neq \mathtt{null}}{\gamma(H,l{\cdot}\mathtt{x}) \triangleq \mathtt{undefined}} \qquad \frac{\pi(H,l,\mathtt{x}) = l' \quad l \neq \mathtt{null}}{\gamma(H,l{\cdot}\mathtt{x}) \triangleq H(l',\mathtt{x})}$$

Operational semantics: $H, L, \mathtt{e} \longrightarrow H', v$

Dereferencing notation: $H, L, \mathtt{e} \xrightarrow{\gamma} H', v \triangleq \exists r.(H, L, \mathtt{e} \longrightarrow H', r \wedge \gamma(H', r) = v)$.

(Value)
$$H, L, \mathtt{v} \longrightarrow H, v$$

(Variable declaration)
$$H, L, \mathtt{var\ x} \longrightarrow H, \mathtt{undefined}$$

(Object creation)
$$\frac{H_0 = H \uplus \mathsf{obj}(l, l_{op}) \quad \forall i \in 1..n. \left(\begin{array}{l} H_{i-1}, L, \mathtt{e_i} \xrightarrow{\gamma} H_i', v_i \\ H_i = H_i'[(l, \mathtt{x_i}) \mapsto v_i] \end{array} \right)}{H, L, \{\mathtt{x_1} : \mathtt{e_1}, \ldots, \mathtt{x_n} : \mathtt{e_n}\} \longrightarrow H_n, l}$$

(Sequence)
$$\frac{H, L, \mathtt{e_1} \longrightarrow H'', r' \quad H'', L, \mathtt{e_2} \longrightarrow H', r}{H, L, \mathtt{e_1; e_2} \longrightarrow H', r}$$

(Binary operator)
$$\frac{H, L, \mathtt{e_1} \xrightarrow{\gamma} H'', v_1 \quad H'', L, \mathtt{e_2} \xrightarrow{\gamma} H', v_2 \quad v_1 \oplus v_2 = v}{H, L, \mathtt{e_1} \oplus \mathtt{e_2} \longrightarrow H', v}$$

(Conditional true)
$$\frac{H, L, \mathtt{e_1} \xrightarrow{\gamma} H'', v \quad \mathsf{True}(v) \quad H'', L, \mathtt{e_2} \longrightarrow H', r}{H, L, \mathtt{if(e_1)\{e_2\}else\{e_3\}} \longrightarrow H', r}$$

(Conditional false)
$$\frac{H, L, \mathtt{e_1} \xrightarrow{\gamma} H'', v \quad \mathsf{False}(v) \quad H'', L, \mathtt{e_3} \longrightarrow H', r}{H, L, \mathtt{if(e_1)\{e_2\}else\{e_3\}} \longrightarrow H', r}$$

(While true)
$$\frac{H, L, \mathtt{e_1} \xrightarrow{\gamma} H'', v \quad \mathsf{True}(v) \quad H'', L, e_2; \mathtt{while(e_1)\{e_2\}} \longrightarrow H', v''}{H, L, \mathtt{while(e_1)\{e_2\}} \longrightarrow H', \mathtt{undefined}}$$

(While false)
$$\frac{H, L, \mathtt{e_1} \xrightarrow{\gamma} H', v \quad \mathsf{False}(v)}{H, L, \mathtt{while(e_1)\{e_2\}} \longrightarrow H', \mathtt{undefined}}$$

(Variable)
$$\frac{\sigma(H, L, x) = l' \quad l' \neq \mathtt{null}}{H, L, \mathtt{x} \longrightarrow H, l'{\cdot}x}$$

(Member access)
$$\frac{H, L, \mathtt{e} \xrightarrow{\gamma} H', l' \quad l' \neq \mathtt{null}}{H, L, \mathtt{e.x} \longrightarrow H', l'{\cdot}x}$$

(Computed member access)
$$\frac{H, L, \mathtt{e_1} \xrightarrow{\gamma} H'', l' \quad l' \neq \mathtt{null} \quad H'', L, \mathtt{e_2} \longrightarrow H', x}{H, L, \mathtt{e_1[e_2]} \longrightarrow H', l'{\cdot}x}$$

(Assignment)
$$\frac{H, L, \mathtt{e_1} \longrightarrow H_1, l{\cdot}x \qquad \mathsf{RW}(H_1, l) \quad H_1, L, \mathtt{e_2} \xrightarrow{\gamma} H_2, v \quad H' = H_2[(l, x) \mapsto v]}{H, L, \mathtt{e_1} = \mathtt{e_2} \longrightarrow H', v}$$

(Function creation)

$$\frac{H' = H \uplus \mathsf{obj}(l, l_{op}) \uplus \mathsf{fun}(l', L, \mathtt{x}, \mathtt{e}, l)}{H, L, \mathtt{function}\ (\mathtt{x})\{\mathtt{e}\} \longrightarrow H', l'}$$

(Named function creation)

$$\frac{H' = H \uplus \mathsf{obj}(l, l_{op}) \uplus \mathsf{fun}(l', l_1{:}L, \mathtt{x}, \mathtt{e}, l) \uplus l_1 \mapsto \{@proto : \mathtt{null}, y : l'\}}{H, L, \mathtt{function}\ \mathtt{y}(\mathtt{x})\{\mathtt{e}\} \longrightarrow H', l'}$$

(This)

$$\frac{\sigma(H, L, @this) = l \quad (l, @this) \mapsto l'}{H, L, \mathtt{this} \longrightarrow H, l'}$$

(Function call)

$$\frac{\begin{array}{l} H, L, \mathtt{e}_1 \longrightarrow H_1, r_1 \qquad \mathsf{This}(H_1, r_1) = l_2 \qquad \gamma(H_1, r_1) = l_1 \\ H_1(l_1, @body) = \lambda \mathtt{x}.\mathtt{e}_3 \qquad H_1(l_1, @scope) = L' \\ H_1, L, \mathtt{e}_2 \xrightarrow{\gamma} H_2, v \\ H_3 = H_2 \uplus \mathsf{act}(l, \mathtt{x}, v, \mathtt{e}_3, l_2) \\ H_3, l{:}L', \mathtt{e}_3 \xrightarrow{\gamma} H', v' \end{array}}{H, L, \mathtt{e}_1(\mathtt{e}_2) \longrightarrow H', v'}$$

(Object construction)

$$\frac{\begin{array}{l} H, L, \mathtt{e}_1 \xrightarrow{\gamma} H_1, l_1 \qquad l_1 \neq \mathtt{null} \qquad H_1(l_1, @body) = \lambda \mathtt{x}.\mathtt{e}_3 \\ H_1(l_1, @scope) = L' \qquad H_1(l_1, \mathtt{prototype}) = v \\ H_1, L, \mathtt{e}_2 \xrightarrow{\gamma} H_2, v_1 \qquad l_2 = \mathsf{SelectProto}(v) \\ H_3 = H_2 \uplus \mathsf{obj}(l_3, l_2) \uplus \mathsf{act}(l, \mathtt{x}, v_1, \mathtt{e}_3, l_3) \\ H_3, l{:}L', \mathtt{e}_3 \xrightarrow{\gamma} H', v_2 \qquad \mathsf{getBase}(l_3, v_2) = l' \end{array}}{H, L, \mathtt{new}\ \mathtt{e}_1(\mathtt{e}_2) \longrightarrow H', l'}$$

(Restricted evaluation)

$$\frac{\begin{array}{l} H, L, \mathtt{e}_1 \xrightarrow{\gamma} H_1, v \qquad \mathtt{e}_3 = \mathsf{parse}(v) \\ H_1, L, \mathtt{e}_2 \xrightarrow{\gamma} H_2, l \\ H_3 = H_2 \uplus l' \mapsto \{@this : l, @proto : \mathtt{null}\} \uplus \mathsf{defs}(\_, l', \mathtt{e}_3) \\ H_3, l'{:}[l], \mathtt{e}_3 \xrightarrow{\gamma} H', v \end{array}}{H, L, \mathtt{reval}(\mathtt{e}_1, \mathtt{e}_2) \longrightarrow H', v}$$

(Freeze)

$$\frac{H, L, \mathtt{e} \xrightarrow{\gamma} H'', l \quad H' = H''[(l, @frozen) \mapsto \mathtt{true}]}{H, L, \mathtt{freeze}(\mathtt{e}) \longrightarrow H', l}$$

(Recursive freeze)

$$\frac{H, L, \mathtt{e} \xrightarrow{\gamma} H'', l \quad H' = H''[\mathsf{auxDef}(H'', l, \{\})]}{H, L, \mathtt{def}(\mathtt{e}) \longrightarrow H', l}$$

Appendix C

Program Logic

Notation: Sorts and Constants

$\epsilon \in \mathcal{X}^{\mathrm{L}} \rightharpoonup \mathcal{V}^{\mathrm{L}}$	Logical environment.
$\mathrm{v} \in \mathcal{V}^{\mathrm{L}}$	Logical values.
$\mathrm{X} \in \mathcal{X}^{\mathrm{L}}$	Logical variables.
E	Logical expressions.
$[\![E]\!]^L_\epsilon$	Logical evaluation.
$av \in \mathcal{V}^{\mathrm{A}} \triangleq \mathcal{V} \cup \{\varnothing\}$	Abstract values.
$h \in (\mathcal{L} \times \mathcal{X}) \rightharpoonup \mathcal{V}^{\mathrm{A}}$	Abstract heap.
$\lfloor h \rfloor (l, x)$	Heap evaluation.
P	Logical assertion.
SET	Generic set.
$h, h_g, L, \epsilon \models P$	Satisfaction relation.
$\{P\}\mathtt{e}\{Q\}$	Hoare triple.

Abstract Values and Abstract Heap.

$av \in \mathcal{V}^{\mathrm{A}} ::= v \mid \varnothing \qquad h : (\mathcal{L} \times \mathcal{X}) \rightharpoonup \mathcal{V}^{\mathrm{A}}$

$\lfloor h \rfloor (l, x) \triangleq h(l, x)$ iff $(l, x) \in \mathsf{dom}(h) \wedge h(l, x) \neq \varnothing$

Logical Expressions and Evaluation: $[\![E]\!]^L_\epsilon$.

$\mathrm{v} \in \mathcal{V}^{\mathrm{L}} ::= e \mid l{\cdot}x \mid av \mid L \qquad \epsilon : \mathcal{X}^{\mathrm{L}} \rightharpoonup \mathcal{V}^{\mathrm{L}}$

$E ::=$	X	Logical variables
	$\mid \mathbf{l}$	Scope list
	$\mid \mathrm{v}$	Logical values
	$\mid E \oplus E$	Binary Operators
	$\mid E{:}E$	List cons
	$\mid E{\cdot}E$	Reference construction
	$\mid \lambda E.E$	Lambda values

$[\![\mathrm{v}]\!]^L_\epsilon \triangleq \mathrm{v} \qquad [\![\mathbf{l}]\!]^L_\epsilon \triangleq L \qquad [\![\mathrm{X}]\!]^L_\epsilon \triangleq \epsilon(\mathrm{X})$

$$\frac{[\![E_2]\!]^L_\epsilon = Ls}{[\![E_1{:}E_2]\!]^L_\epsilon \triangleq [\![E_1]\!]^L_\epsilon{:}Ls} \qquad \frac{[\![E_1]\!]^L_\epsilon = l' \wedge [\![E_2]\!]^L_\epsilon = \mathtt{x}}{[\![E_1{\cdot}E_2]\!]^L_\epsilon \triangleq [\![E_1]\!]^L_\epsilon{\cdot}[\![E_2]\!]^L_\epsilon}$$

$$\frac{[\![E_1]\!]^L_\epsilon = v \wedge [\![E_2]\!]^L_\epsilon = v'}{[\![E_1 \oplus E_2]\!]^L_\epsilon \triangleq v \mathbin{\overline{\oplus}} v'} \qquad \frac{[\![E_1]\!]^L_\epsilon = \mathtt{x}}{[\![\lambda E_1.E_2]\!]^L_\epsilon \triangleq \lambda [\![E_1]\!]^L_\epsilon.[\![E_2]\!]^L_\epsilon}$$

Assertions

$P ::=$	$P \wedge P \mid P \vee P \mid \neg P \mid \text{true} \mid \text{false}$	Boolean formulas
$\mid$	$P * P \mid P \mathrel{-\!\!*} P \mid P \mathbin{⋉} P$	Structural formulas
$\mid$	$\circledast_{x \in \text{SET}} P(x)$	Iterative $*$
$\mid$	$P \mathrel{-\!\!\circledast} Q$	Global 'hypothesis' formula
$\mid$	$(E, E) \mapsto E \mid \oslash$	Heap formulas
$\mid$	$E \leftarrow E$	Backpointer formula
$\mid$	$E = E$	Expression equality
$\mid$	$\forall X.\, P \mid \exists X.\, P$	First-order formulas

Assertion satisfaction relation

$$
\begin{array}{lcl}
h, h_g, L, \epsilon \models \text{true} & & \\
h, h_g, L, \epsilon \models P \wedge Q & \iff & (h, h_g, L, \epsilon \models P) \wedge (h, h_g, L, \epsilon \models Q) \\
h, h_g, L, \epsilon \models P \vee Q & \iff & (h, h_g, L, \epsilon \models P) \vee (h, h_g, L, \epsilon \models Q) \\
h, h_g, L, \epsilon \models \neg P & \iff & \neg (h, h_g, L, \epsilon \models P) \\
h, h_g, L, \epsilon \models P * Q & \iff & \exists h_1, h_2.\, h \equiv h_1 \uplus h_2 \wedge \\
 & & \quad (h_1, h_g, L, \epsilon \models P) \wedge (h_2, h_g, L, \epsilon \models Q) \\
h, h_g, L, \epsilon \models P ⋉ Q & \iff & \exists h_1, h_2, h_3.\, h \equiv h_1 \uplus h_2 \uplus h_3 \wedge \\
 & & \quad (h_1 \uplus h_3, h_g, L, \epsilon \models P) \wedge (h_2 \uplus h_3, h_g, L, \epsilon \models Q) \\
h, h_g, L, \epsilon \models P \mathrel{-\!\!*} Q & \iff & \forall h'.\, (h', h_g, L, \epsilon \models P) \wedge h \# h' \wedge h' \subseteq h_g \\
 & & \quad \Rightarrow (h \uplus h', h_g, L, \epsilon \models Q) \\
h, h_g, L, \epsilon \models P \mathrel{-\!\!\circledast} Q & \iff & \forall h'.\, (h', h_g[h'], L, \epsilon \models P) \wedge h \# h' \\
 & & \quad \Rightarrow (h \uplus h', h_g[h'], L, \epsilon \models Q) \\
h, h_g, L, \epsilon \models \circledast_{x \in \{\}} P(x) & \iff & h, h_g, L, \epsilon \models \oslash \\
h, h_g, L, \epsilon \models \circledast_{x \in \text{SET}} P(x) & \iff & y \in \text{SET} \wedge h, h_g, L, \epsilon \models P(y) * (\circledast_{x \in (\text{SET} \setminus y)} P(x)) \\
h, h_g, L, \epsilon \models \oslash & \iff & h = \mathtt{emp} \\
h, h_g, L, \epsilon \models (E_1, E_2) \mapsto E_3 & \iff & h \equiv ([\![E_1]\!]^L_\epsilon, [\![E_2]\!]^L_\epsilon) \mapsto [\![E_3]\!]^L_\epsilon \\
h, h_g, L, \epsilon \models E_1 \leftarrow E_2 & \iff & h \equiv ([\![E_1]\!]^L_\epsilon, @bp) \mapsto v \wedge \forall (l, x) \in \mathsf{dom}(h_g).\, h_g(l, x) = [\![E_1]\!]^L_\epsilon \\
 & & \quad \Rightarrow (l, x) \in [\![E_2]\!]^L_\epsilon \\
h, h_g, L, \epsilon \models E_1 = E_2 & \iff & [\![E_1]\!]^L_\epsilon = [\![E_2]\!]^L_\epsilon \\
h, h_g, L, \epsilon \models \exists \mathrm{X}.\, P & \iff & \exists \mathrm{v}.\, h, h_g, L, [\epsilon | \mathrm{X} \leftarrow \mathrm{v}] \models P \\
h, h_g, L, \epsilon \models \forall \mathrm{X}.\, P & \iff & \forall \mathrm{v}.\, h, h_g, L, [\epsilon | \mathrm{X} \leftarrow \mathrm{v}] \models P
\end{array}
$$

Definition of Hoare triple and soundness properties

$$\{P\}\mathsf{e}\{Q\}$$

Soundness:

$$(h, h \uplus h_f, L, (\epsilon \setminus \mathbf{r}) \models P) \wedge h, L, \mathsf{e} \rightsquigarrow h', v \Rightarrow (h', h' \uplus h_f, L, [\epsilon | \mathbf{r} \leftarrow v] \models Q)$$

Fault avoidance:

$$h, h_g, L, (\epsilon \setminus \mathbf{r}) \models P \wedge h \subseteq h_g \Rightarrow h, L, \mathsf{e} \not\rightsquigarrow \mathsf{fault}$$

Safety monotonicity:

$$(h, h_g, L, \epsilon \models P) \wedge h \# h' \wedge h, L, \mathsf{e} \not\rightsquigarrow \mathsf{fault} \wedge h \subseteq h_g \Rightarrow h \uplus h', L, \mathsf{e} \not\rightsquigarrow \mathsf{fault}$$

Frame property:

$$(h, h_g, L, \epsilon \models P) \wedge h, L, \mathsf{e} \not\rightsquigarrow \mathsf{fault} \wedge h \uplus h', L, \mathsf{e} \rightsquigarrow h'_2 \wedge h \subseteq h_g \Rightarrow h, L, \mathsf{e} \rightsquigarrow h_2 \wedge h'_2 = h_2 \uplus h'$$

Auxiliary Predicates

$\mathsf{obj}_E(E_1 : E'_1, ..., E_n : E'_n) \triangleq (E, E_1) \mapsto E'_1 * ... * (E, E_n) \mapsto E'_n$
$\mathsf{newobj}_L(V_1, ..., V_n) \triangleq \circledast_{V \in \mathcal{X} \setminus \{V_1, ..., V_n\}} (L, V) \mapsto \varnothing$
$\mathsf{fullobj}_E(E_1 : E'_1, \ldots, E_n : E'_n) \triangleq \mathsf{obj}_E(E_1 : E'_1, \ldots, E_n : E'_n) * \mathsf{newobj}_E(E_1, \ldots, E_n)$
$\mathsf{fun}_F(E_1, E_2, E_3, E_4) \triangleq (F, @scope) \mapsto E_1 * (F, @body) \mapsto \lambda E_2.E_3 * (F, \texttt{prototype}) \mapsto E_4 * (F, @proto) \mapsto l_{fp}$
$\mathsf{newfun}_E(E_1, E_2, E_3, E_4) \triangleq \mathsf{fun}_E(E_1, E_2, E_3, E_4) * \mathsf{newobj}_E(@proto, \texttt{prototype}, @scope, @body)$
$\mathsf{scopeBpsUpd}(Ls, s, s', n) \triangleq \circledast_{0 \leq i < \mathrm{length}(Ls)} (\mathrm{item}(i, Ls) \leftarrow \mathrm{item}(i, s') * \mathrm{item}(i, s') \doteq \mathrm{item}(i, s) \cup n)$
$\mathsf{scopeBps}(Ls, s) \triangleq \mathsf{scopeBpsUpd}(Ls, s, s, \{\})$
$\beta(V, \_, \_, \_) \triangleq V \notdot \mathcal{L}$
$\beta(V, L, x, s) \triangleq V \doteq \mathcal{L} * V \leftarrow \{(L, x)\} \cup s$
$\mathsf{auxDefGet}(V, s) \triangleq V \notdot \mathcal{L} \vee V \doteq s$
$\mathsf{auxDefGet}(V, s) \triangleq V \notdot s \wedge (V, @frozen) \mapsto \_ * (\maltese_{x_n \in \mathcal{X}^{\mathrm{v}}} (V, x_n) \mapsto V' \maltese \mathsf{auxDefGet}(V', s \cup \{V\}))$
$\mathsf{auxDefSet}(V, s, b) \triangleq V \notdot \mathcal{L} \vee V \doteq s$
$\mathsf{auxDefSet}(V, s, b) \triangleq V \notdot s \wedge (V, @frozen) \mapsto b * (\maltese_{x_n \in \mathcal{X}^{\mathrm{v}}} (V, x_n) \mapsto V' \maltese \mathsf{auxDefSet}(V', s \cup \{V\}, b))$
$\mathsf{RW}(L) \triangleq (L, @frozen) \mapsto \texttt{false}$
$\mathsf{decls}(X, L, \mathtt{e}) \triangleq \mathtt{x_1}, ..., \mathtt{x_n} \quad \text{where } (L, \mathtt{x_i}) \in \mathsf{dom}(\mathsf{defs}(X, L, \mathtt{e}))$
$\mathsf{This}(L \cdot \_, L) \triangleq (L, @this) \mapsto \varnothing$
$\mathsf{This}(L \cdot \_, \texttt{undefined}) \triangleq \exists V. (L, @this) \mapsto V * V \neq \varnothing$

Local Variable Definitions

$\mathsf{defs}(X, L, \texttt{var y}) \triangleq (L, \mathtt{y}) \mapsto \texttt{undefined} * X \neq \mathtt{y}$
$\mathsf{defs}(X, L, \mathtt{e_1} = \mathtt{e_2}) \triangleq \mathsf{defs}(X, L, \mathtt{e_1})$
$\mathsf{defs}(X, L, \mathtt{e_1; e_2}) \triangleq \mathsf{defs}(X, L, \mathtt{e_1}) \maltese \mathsf{defs}(X, L, \mathtt{e_2})$
$\mathsf{defs}(X, L, \texttt{if}(\mathtt{e_1})\{\mathtt{e_2}\}\{\mathtt{e_3}\}) \triangleq \mathsf{defs}(X, L, \mathtt{e_2}) \maltese \mathsf{defs}(X, L, \mathtt{e_3})$
$\mathsf{defs}(X, L, \texttt{while}(\mathtt{e_1})\{\mathtt{e_2}\}) \triangleq \mathsf{defs}(X, L, \mathtt{e_2})$
$\mathsf{defs}(X, L, \mathtt{e}) \triangleq \oslash \quad \text{otherwise}$

Inference Rules

(Declaration)
$\{\oslash\}\texttt{var x}\{\mathbf{r} \doteq \texttt{undefined}\}$

(Value)
$\{\oslash\}\mathtt{v}\{\mathbf{r} \doteq \mathtt{v}\}$

(Variable)
$$\frac{P = \sigma(Ls_1, \mathbf{l}, \mathtt{x}, L) \maltese \gamma(Ls_2, L \cdot \mathtt{x}, V) * L \neq \texttt{null}}{\{P\}\mathtt{x}\{P * \mathbf{r} \doteq L \cdot \mathtt{x}\}}$$

(Member Access)
$$\frac{\{P\}\mathtt{e}\{Q * \mathbf{r} \doteq V\} \quad Q = R * \gamma(Ls, V, L) * L \neq \texttt{null} * L \doteq \mathcal{L}}{\{P\}\mathtt{e.x}\{Q * \mathbf{r} \doteq L \cdot \mathtt{x}\}}$$

(Computed Access)
$$\frac{\{P\}\mathtt{e_1}\{R * \mathbf{r} \doteq V_1\} \quad R = S_1 * \gamma(Ls_1, V_1, L) * L \neq \texttt{null} * L \doteq \mathcal{L} \quad \{R\}\mathtt{e_2}\{Q * X \doteq \mathcal{X}^{\mathrm{u}} * \mathbf{r} \doteq V_2\} \quad Q = S_2 * \gamma(Ls_2, V_2, X)}{\{P\}\mathtt{e_1}[\mathtt{e_2}]\{Q * \mathbf{r} \doteq L \cdot X\}}$$

(Bin Op)
$$\frac{\{P\}\mathtt{e_1}\{R * \mathbf{r} \doteq V_1\} \quad R = S_1 * \gamma(Ls_1, V_1, V_3) \quad \{R\}\mathtt{e_2}\{Q * \mathbf{r} \doteq V_2\} \quad Q = S_2 * \gamma(Ls_2, V_2, V_4) \quad V = V_3 \mathbin{\bar{\oplus}} V_4}{\{P\}\mathtt{e_1} \oplus \mathtt{e_2}\{Q * \mathbf{r} \doteq V\}}$$

(Assign)
$$\frac{\{P\}\mathtt{e_1}\{R * \mathbf{r} \doteq L \cdot X\} \quad \{R\}\mathtt{e_2}\{Q * (L, X) \mapsto V_3 * \mathbf{r} \doteq V_1\} \quad Q = S * \gamma(Ls, V_1, V_2) * \mathsf{RW}(L) * \beta(V_2, L, X, s)}{\{P\}\mathtt{e_1} = \mathtt{e_2}\{Q * (L, X) \mapsto V_2 * \mathbf{r} \doteq V_2\}}$$

(This)

$$\frac{P = \sigma(Ls_1, \mathbf{l}, @this, L_1) \bowtie \pi(Ls_2, L_1, @this, L_2) \bowtie (L_2, @this) \mapsto V}{\{P\}\texttt{this}\{P * \mathbf{r} \doteq V\}}$$

(Function)

$$\frac{\begin{array}{l} P = l_{op} \leftarrow s_1 * l_{fp} \leftarrow s_2 * \mathsf{scopeBps}(\mathbf{l}, ss) \\ Q = \exists L_1, L_2. \left(\begin{array}{l} \mathsf{fullobj}_{L_1}(@proto : l_{op}) * L_1 \leftarrow \{(L_2, \texttt{prototype})\} * l_{op} \leftarrow s_1 \cup \{(L_1, @proto)\} * \\ \mathsf{newfun}_{L_2}(\mathbf{l}, \mathtt{x}, \mathtt{e}, L_1) * L_2 \leftarrow \{\} * l_{fp} \leftarrow s_2 \cup \{(L_2, @proto)\} * \\ \mathbf{r} \doteq L_2 * \mathsf{scopeBpsUpd}(\mathbf{l}, ss, ss', \{(L_2, @scope)\}) \end{array} \right) \end{array}}{\{P\}\texttt{function}\ (\mathtt{x})\{\mathtt{e}\}\{Q\}}$$

(Named Function)

$$\frac{\begin{array}{l} P = l_{op} \leftarrow s_1 * l_{fp} \leftarrow s_2 * \mathsf{scopeBps}(\mathbf{l}, ss) \\ Q = \exists L_1, L_2, L_3. \left(\begin{array}{l} \mathsf{fullobj}_{L_1}(@proto : l_{op}) * L_1 \leftarrow \{(L_2, \texttt{prototype})\} * l_{op} \leftarrow s_1 \cup \{(L_1, @proto)\} * \\ \mathsf{newfun}_{L_2}((L_3 : \mathbf{l}), \mathtt{x}, \mathtt{e}, L_1) * L_2 \leftarrow \{(L_3, \mathtt{y})\} * l_{fp} \leftarrow s_2 \cup \{(L_2, @proto)\} * \\ \mathsf{fullobj}_{L_3}(@proto : \texttt{null}, \mathtt{y} : L_2) * L_3 \leftarrow \{(L_2, @scope)\} * \\ \mathbf{r} \doteq L_2 * \mathsf{scopeBpsUpd}(\mathbf{l}, ss, ss', \{(L_2, @scope)\}) \end{array} \right) \end{array}}{\{P\}\texttt{function}\ \mathtt{y}(\mathtt{x})\{\mathtt{e}\}\{Q\}}$$

(Object)

$$\frac{\begin{array}{l} \forall i \in 1..n. \left(\begin{array}{l} P_i = R_i * \gamma(Ls_i, Y_i, X_i) * X_i \leftarrow s_i \\ \{P_{i-1}\}\mathtt{e_i}\{P_i * \mathbf{r} \doteq Y_i\} \end{array} \right) \\ P_n = R * l_{op} \leftarrow s_{op} \\ Q = R * \exists L. \left(\begin{array}{l} \mathsf{newobj}_L(@proto, \mathtt{x_1}, ..., \mathtt{x_n}) * \\ \circledast_{1 \le i \le n}((L, \mathtt{x_i}) \mapsto X_i * \beta(X_i, L, \mathtt{x_i}, s_i)) * \\ (L, @proto) \mapsto l_{op} * l_{op} \leftarrow s_{op} \cup \{(L, @proto)\} * \\ \mathbf{r} \doteq L * L \leftarrow \{\} \end{array} \right) \\ \mathtt{x_1} \neq \cdots \neq \mathtt{x_n} \qquad \mathbf{r} \notin \mathsf{fv}(P_n) \end{array}}{\{P_0\}\{\mathtt{x_1} : \mathtt{e_1}, ..., \mathtt{x_n} : \mathtt{e_n}\}\{Q\}}$$

(Function Call)

$$\frac{\begin{array}{l} \{P\}\mathtt{e_1}\{R_1 * \mathbf{r} \doteq F_1\} \\ R_1 = \left(\begin{array}{l} S_1 \bowtie \mathsf{This}(F_1, T) \bowtie \gamma(Ls_1, F_1, F_2) * \\ (F_2, @body) \mapsto \lambda X.\mathtt{e_3} * (F_2, @scope) \mapsto Ls_2 \end{array} \right) \\ \{R_1\}\mathtt{e_2}\{R_2 * \beta(T, s) * \beta(V_2, s') * \mathbf{l} \doteq Ls_3 * \mathbf{r} \doteq V_1\} \\ R_2 = S_2 * \gamma(Ls_4, V_1, V_2) \\ R_3 = R_2 * \exists L. \left(\begin{array}{l} (L, X) \mapsto V_2 * \beta(V_2, L, X, s') * \\ (L, @this) \mapsto T * \beta(T, L, @this, s) * (L, @proto) \mapsto \texttt{null} * \\ \mathsf{defs}(X, L, \mathtt{e_3}) * \mathsf{newobj}_L(@proto, @this, \mathtt{x}, \mathsf{decls}(X, L, \mathtt{e_3})) * \\ L \leftarrow \{\} * \mathbf{l} \doteq L : Ls_2 \end{array} \right) \\ \{R_3\}\mathtt{e_3}\{\exists L. Q * \mathbf{l} \doteq L : Ls_2\} \qquad \mathbf{l} \notin \mathsf{fv}(Q) \cup \mathsf{fv}(R_2) \end{array}}{\{P\}\mathtt{e_1}(\mathtt{e_2})\{\exists L. Q * \mathbf{l} \doteq Ls_3\}}$$

(New) —

(Restricted evaluation)

$$\frac{\begin{array}{l} \{P\}\mathtt{e_1}\{R_1 * \mathbf{r} \doteq V_1\} \\ R_1 = S_1 * \gamma(\_, V_1, V_2) * V_2 \dot{\in} \mathcal{M} \\ \mathsf{parse}(V_2) = \mathtt{e_3} \\ \{R_1\}\mathtt{e_2}\{R_2 * \mathbf{r} \doteq V_3 * \mathbf{l} \doteq Ls\} \\ R_2 = \left(\begin{array}{l} S_2 * \gamma(\_, V_3, V_4) * V_4 \dot{\in} \mathcal{L} * V_4 \leftarrow s \cup \{(L, @this)\} * \\ \exists L. R_3 * \mathsf{newobj}_L(@proto, @this, \mathsf{decls}(\_, L, \mathtt{e_3})) * \\ \mathsf{obj}_L(@this : V_4, @proto : \texttt{null}) * \mathsf{defs}(\_, L, \mathtt{e_3}) \end{array} \right) \\ R_3 = (\mathbf{l} \doteq L : [V_4]) \\ \{R_2\}\mathtt{e_3}\{\exists L. Q * R_3\} \end{array}}{\{P\}\texttt{reval}(\mathtt{e_1}, \mathtt{e_2})\{\exists L. Q * \mathbf{l} \doteq Ls\}}$$

(Freeze)

$$\frac{\begin{array}{l} \{P\}\mathtt{e}\{Q * \mathbf{r} \doteq V_1 * (V_2, @frozen) \mapsto V_3\} \\ Q = \gamma(Ls, V_1, V_2) * S \end{array}}{\{P\}\texttt{freeze}(\mathtt{e})\{Q * (V_2, @frozen) \mapsto \texttt{true} * \mathbf{r} \doteq V_2\}}$$

(Recursive freeze)

$$\frac{\{P\}\mathtt{e}\{Q * \mathbf{r} \doteq V_2 * \mathsf{auxDefGet}(V_2, \{\})\} \quad Q = \gamma(Ls, V_1, V_2) * S}{\{P\}\mathtt{def}(\mathtt{e})\{Q * \mathbf{r} \doteq V_2 * \mathsf{auxDefSet}(V_2, \{\}, \mathtt{true})\}}$$

(Sequence)

$$\frac{\{P\}\mathtt{e}_1\{R\} \quad \{R\}\mathtt{e}_2\{Q\}}{\{P\}\mathtt{e}_1;\mathtt{e}_2\{Q\}}$$

(If True)

$$\frac{\{P\}\mathtt{e}_1\{S * \mathsf{True}(V_2) * \mathbf{r} \doteq V_1\} \qquad S = R * \gamma(Ls, V_1, V_2) \quad \{S\}\mathtt{e}_2\{Q\}}{\{P\}\mathtt{if}(\mathtt{e}_1)\{\mathtt{e}_2\}\mathtt{else}\{\mathtt{e}_3\}\{Q\}}$$

(If False)

$$\frac{\{P\}\mathtt{e}_1\{S * \mathsf{False}(V_2) * \mathbf{r} \doteq V_1\} \qquad S = R * \gamma(Ls, V_1, V_2) \quad \{S\}\mathtt{e}_3\{Q\}}{\{P\}\mathtt{if}(\mathtt{e}_1)\{\mathtt{e}_2\}\mathtt{else}\{\mathtt{e}_3\}\{Q\}}$$

(While)

$$\frac{\{P\}\mathtt{e}_1\{S * \mathbf{r} \doteq V_1\} \qquad S = R * \gamma(Ls, V_1, V_2) \quad \{S * \mathsf{True}(V_2)\}\mathtt{e}_2\{P\} \quad Q = S * \mathsf{False}(V_2) * \mathbf{r} \doteq \mathtt{undefined} \qquad \mathbf{r} \notin \mathsf{fv}(R)}{\{P\}\mathtt{while}(\mathtt{e}_1)\{\mathtt{e}_2\}\{Q\}}$$

(Frame)

$$\frac{\{P\}\mathtt{e}\{Q\}}{\{P * R\}\mathtt{e}\{Q * R\}}$$

(Consequence)

$$\frac{\{P'\}\mathtt{e}\{Q'\} \quad P \Rightarrow P' \quad Q' \Rightarrow Q}{\{P\}\mathtt{e}\{Q\}}$$

(Elimination)

$$\frac{\{P\}\mathtt{e}\{Q\}}{\{\exists X.\, P\}\mathtt{e}\{\exists X.\, Q\}}$$

(Disjunction)

$$\frac{\{P_1\}\mathtt{e}\{Q_1\} \quad \{P_2\}\mathtt{e}\{Q_2\}}{\{P_1 \vee P_2\}\mathtt{e}\{Q_1 \vee Q_2\}}$$