A Sound Type System for the Meta Language of the JavaScript Standard

Pedro José Fernandes Nunes

Thesis to obtain the Master of Science Degree in

Applied Mathematics

Supervisor(s): Prof. Paulo Mateus
Prof. José Fragoso Santos

Examination Committee

Chairperson: Pedro Resende
Supervisor: José Fragoso Santos
Member of the Committee: Ana Matos

December 2021

Acknowledgments

I would like to thank both my supervisors, especially, Prof. José Fragoso Santos, for all the weekends spent helping me. If not for his support this thesis would certainly not have been concluded.

Also, I would like to thank my parents for their huge emotional and financial support and my friends for their humour which kept me cheered up.

Last but not least, I would also like to thank my boss for all the flexibility provided, enabling me to successfully complete my thesis.

Resumo

JavaScript é a linguagem de programação mais utilizada para scripting do lado do cliente na world wide web e tem também ganho popularidade em outros tipos de aplicações através de Node.js. A complexidade da semântica do JavaScript torna-a um alvo difícil para análise estática. Consequentemente e de forma a auxiliar a análise e especificação de programas em JavaScript, foi desenvolvida uma nova linguagem intermédia não tipada denominada ECMA-SL. Nesta tese introduzimos a Typed ECMA-SL, uma versão tipada da ECMA-SL, juntamente com um sistema de tipos sensível ao fluxo para a linguagem. Definimos ainda duas semânticas operacionais, uma de grande passo e outra de pequeno passo, para a Typed ECMA-SL e provámos a correção do sistema de tipos proposto com respeito a ambas as semânticas.

Palavras-chave: JavaScript, Sistemas de Tipos, Sensibilidade ao Fluxo, Correção de Tipos

Abstract

JavaScript is the programming language most commonly used for client-side scripting in the world wide web and has been gaining popularity for other types of applications via Node.js. The complexity of the JavaScript semantics makes it a hard target for static analyses. Thus, a new intermediate untyped language named ECMA-SL was developed to assist with the analysis and specification of JavaScript programs. In this thesis, we introduce Typed ECMA-SL, a typed version of ECMA-SL, together with a flow-sensitive type system for the language. We further define a big-step and a small-step operational semantics for Typed ECMA-SL and prove the soundness of the proposed type system with respect to both semantics.

Keywords: JavaScript, Type Systems, Flow-sensitivity, Type Soundness

Contents

List of Figures

Chapter 1

Introduction

JavaScript is the programming language most commonly used for client-side scripting in the world wide web [1] and has been gaining increasingly popularity for other types of applications via Node.js [2], a run-time environment for developing stand-alone JavaScript applications built on top of the V8 JavaScript engine [3]. In order to guarantee that JavaScript programs behave consistently throughout all existing browsers, the Ecma International association develops and maintains the JavaScript standard, a long complex document written in English, about one thousand pages long, describing both the syntax and semantics of the JavaScript language. The complexity of the JavaScript semantics makes it a hard target for static analyses, which, in order to be sound, have to reason about all the corner cases described in the official standard.

The standard way to deal with the complexity of real world programming languages when designing new program analyses is to first compile the given program to a simpler intermediate language and then apply the analysis at the intermediate language level. Following this approach, a research team at INESC-ID developed ECMA-SL, a new intermediate language for JavaScript analysis and specification. The ECMA-SL project [4] comes with a compiler from JavaScript to ECMA-SL, thereby allowing new static analyses for JavaScript to target ECMA-SL instead of JavaScript directly. ECMA-SL is a simple untyped imperative language with extensible objects and standard control flow constructs. In contrast to the semantics of JavaScript which is about 1000 pages long, the semantics of ECMA-SL can be formally described in one page, making it a suitable target for static analysis.

Currently the ECMA-SL project supports ECMAScript 5, the 5th version [5] of the JavaScript standard, which is now in its 12th version [6] (ECMAScript 12). The ECMA-SL project has at its core an ECMAScript 5 interpreter written in ECMA-SL called ECMARef5 [7], which consists of more than 10K lines of ECMA-SL code. In order to adapt the ECMA-SL project to the more recent versions of the ECMAScript standard, one has to adapt and extend the ECMARef5 interpreter. This is by no means an easy task as the size and complexity of the standard grew substantially since its 5th version. Furthermore, the fact that ECMA-SL is untyped makes any sort of refactoring of the existing code base extremely error prone and time consuming. For this reason, the aim of this project is to streamline the management and maintenance of the ECMARef interpreter by adding a type system to ECMA-SL.

A type system is a syntactic method for checking the absence of certain classes of errors in programs by classifying the given program's statements and expressions according to the kinds of values that it computes. Typically, a type system is defined as a set of rules, with each rule applying to a specific phrase of the language. These rules target type errors such as an operand or argument passed to a function being incompatible with the type expected by that operator or function. Ideally, type systems are supposed to be *sound*: if a sound type system accepts a program, then there are no inputs for which the execution of that program throws a runtime error. In order to guarantee soundness, type systems have to be conservative, meaning that they have to reject not only incorrect programs but also correct programs that cannot be proven so. We say that a type system is more precise than another if the former rejects fewer correct programs than the latter. In practice, there is a trade-off between precision and complexity of type annotations. The more precise a type system is, the more complex and unwieldy are its corresponding type annotations. Hence, in general, more precise type systems are more difficult to use. In order to avoid this trade-off, some type systems are purposely designed to be unsound with the goal of rejecting as few correct programs as possible whilst not having an overly complex notational burden.

Much like JavaScript, ECMA-SL is a highly dynamic programming language, including features such as extensible objects and dynamic binding of function calls, which make it a hard target for standard type systems. In particular, the combination of aliasing with object mutability is difficult to control if one wants to keep the type system both sound and precise. The goal of this thesis is to formalise a typed version of ECMA-SL, named Typed ECMA-SL, together with a type system with the two following characteristics:

- Soundness: well-typed programs cannot go wrong;
- Flow-Sensitivity: program variables and objects are allowed to change their types during execution.

Flow-sensitivity is key for precision, allowing us to keep our type system as little restrictive as possible. The key idea of our type system is to explicitly track object aliasing and constrain object mutation. In particular, objects are only allowed to be mutated if there is a single pointer to them.

With the gaining popularity of JavaScript, many industrial and academic research groups have developed type systems for different fragments of the language [8–10]. As ECMA-SL and JavaScript have many common features, one would expect that one of the proposed type systems for JavaScript could be applied to ECMA-SL. This is, however, not the case as none of these systems meets our requirements; some of them are explicitly unsound [8], some are flow-insensitive [9], and others are overly complex due to features of JavaScript that are not included in ECMA-SL [10] such as prototype inheritance.

We consider this thesis to have three contributions: first, the formalisation of the Typed ECMA-SL language; second, the development of a type system based on a novel idea - open/closed types for tracking aliasing; finally, two soundness proofs written with respect to two different operational semantics. Below, we briefly describe each of these contributions.

Typed ECMA-SL The first contribution of this work is the definition of Typed ECMA-SL, a typed version of ECMA-SL. In order to facilitate the transition for developers, Typed ECMA-SL was designed to be as similar to ECMA-SL as possible, thus minimizing the number of extra annotations required.

Type System The developed type system is the central contribution of this thesis as it not only provides a set of rules for supporting the development of correct programs, but also presents a novel idea which can be adapted to other object-oriented scripting languages: open/closed objects. This idea, which is at the core of our type system, is key for allowing it to be flow-sensitive, while keeping it sound.

Soundness Proofs We provide two soundness proofs for our type system. In order to do this we introduce two different semantics: a big-step semantics [11] and a small-step semantics [12]. By proving the soundness of our type system we ensure that well-typed programs cannot go wrong, which was one of our type system's requirements. We chose to provide two different proofs to explore different trade-offs between clarity and expressivity.

1.1 Thesis Outline

This thesis starts by introducing, in Chapter 2, Typed ECMA-SL, our typed version of ECMA-SL. In this Chapter we cover the syntax of the language, comparing it against its untyped version and presenting our type system for it. We end this chapter with two examples that help understanding our type system and the ideas behind it. The two following Chapters, Chapter 3 and Chapter 4, are similar in their structure. They start by introducing some preliminary definitions and each semantics in scope (big-step semantics for Chapter 3 and small-step semantics for Chapter 4). Afterwards, we present their respective soundness theorems and associated lemmas. We end both Chapters by extending our semantics and soundness proofs with the function call and return statements. In Chapter 5, we give an overview of the related work, comparing our type system against the existing type systems for JavaScript. Chapter 6 draws some conclusions about our work and points out some future research directions.

Chapter 2

Typed ECMA-SL

ECMA-SL is a simple imperative language with extensible objects developed to assist with JavaScript analysis and specification. The ECMA-SL language was used to develop ECMARef, a new JavaScript reference interpreter that follows the ECMAScript standard [5], the official JavaScript standard, faithfully. The ECMA-SL project has been developed by a research team at INESC-ID and has been thoroughly tested against Test262 [13], the official ECMAScript conformance suite.

So far, ECMA-SL is an untyped language and, therefore, it has two major disadvantages when compared to typed languages. Firstly, a typed language allows for the static detection of code errors. Secondly, programs written in untyped languages are harder to maintain than typed languages, which promote a design-by-contract approach to the software development process, with function signatures acting as a clear interface between the code of the corresponding functions and the programs that use them. This project contributes to the overall ECMA-SL project by designing a new typed version of ECMA-SL, which mitigates these two defects.

In this chapter, we introduce our typed version of ECMA-SL, called Typed ECMA-SL, together with its type system. The chapter is structured into two sections. Section 2.1 describes the syntax of Typed ECMA-SL, highlighting the differences with respect to untyped ECMA-SL. Section 2.2 describes the type system using illustrative examples to explain its main constraints.

2.1 Syntax

A Typed ECMA-SL program $p \in Progs$ is a collection of Typed ECMA-SL functions. A Typed ECMA-SL function $f \in Funcs$ is of the form $function\ f(x_1 : \tau_1, ..., x_n : \tau_n)\{s\}$, where f is the identifier, $x_1, ..., x_n$ are the function formal parameters with types $\tau_1, ..., \tau_n$, and s is the body of the function. The syntax of Typed ECMA-SL is given in Figure 2.1, mostly coinciding with that of untyped ECMA-SL.

Expressions Typed ECMA-SL expressions include literals, variables and a variety of unary and binary operators. Literals might be booleans, strings or numbers.

Expressions

$$
\begin{aligned}
e \in \mathcal{E} ::= {} & x \\
\mid {} & v \\
\mid {} & \oplus(e) \\
\mid {} & \otimes(e_1, e_2)
\end{aligned}
$$

$$\otimes \in \{+, x, -, ...\}$$

$$
\begin{aligned}
v \in \mathcal{V} ::= {} & true \\
\mid {} & false \\
\mid {} & n \in \mathbb{N} \\
\mid {} & string
\end{aligned}
$$

Statements

$$
\begin{aligned}
s \in \mathcal{S} ::= {} & x := e \\
\mid {} & x.p := e \\
\mid {} & x := e.p \\
\mid {} & x := \{\} \\
\mid {} & \mathsf{commit}(x) \\
\mid {} & \mathsf{delete}(x.p) \\
\mid {} & \mathsf{skip} \\
\mid {} & s_1; s_2 \\
\mid {} & \mathsf{if}(e)\{s_1\} \ \mathsf{else} \ \{s_2\} \\
\mid {} & \mathsf{while}(e)\{s\} \\
\mid {} & x := f(e_1, ..., e_n) \\
\mid {} & \mathsf{return}(e)
\end{aligned}
$$

Types

$$
\begin{aligned}
\tau \in T ::= {} & number \\
\mid {} & string \\
\mid {} & boolean \\
\mid {} & null \\
\mid {} & undefined \\
\mid {} & \{p_i : \tau_i|_{i=1}^{n}\}^{\circ} \\
\mid {} & \{p_i : \tau_i|_{i=1}^{n}\}^{\bullet} \\
\mid {} & (\tau_1, ..., \tau_n) \rightarrow \tau
\end{aligned}
$$

Figure 2.1: Typed ECMA-SL Syntax

Statements Typed ECMA-SL statements include:

- the standard control flow statements: if, while, sequence, function call, and return;
- a variable assignment statement;
- statements for interacting with extensible objects: object creation, field lookup, field deletion, and field assignment;
- a special statement commit to provide information to the type system, which will be discussed later in the section.

Typed vs Untyped ECMA-SL The main differences between Typed ECMA-SL and ECMA-SL are the following:

1. Function parameters must be annotated with their corresponding types.
2. The syntax is extended with a special statement `commit` to provide information to the type system.
3. We restrict field look-ups, deletions, and assignments to require the name of the field to appear statically. For instance, we do not support the JavaScript syntax o[x], where x is a program variable that denotes the name of the field being inspected; instead, we only have the syntax o.p, where p is exactly the name of the field being inspected.
4. Functions calls are fully static; that is, the identifier of the function to be called must be known at static time. In contrast, in untyped ECMA-SL, that identifier can be computed dynamically.

Types Typed ECMA-SL includes three main categories of types: primitive types, function types, and object types. Primitive types comprise the string type, the number type, the boolean type, and the special undefined and null types. Function types have their standard interpretation; the type $(\tau_1, ..., \tau_n) \rightarrow \tau$ is the type of the functions that take arguments of types $\tau_1, ..., \tau_n$ and produce a result of type τ. Object types are more complicated. The object type $\{p_i : \tau_i|_{i=1}^{n}\}^*$ denotes objects that only contain the fields p_1 to p_n, mapping each field p_i to a value of type τ_i. Given an object type $\tau = \{p_i : \tau_i|_{i=1}^{n}\}^*$, we write $dom(\tau)$ to mean the set of fields that it contains: $\{p_i \,|_{i=1}^{n}\}$ and $\lfloor\tau\rfloor$ to refer to its openness flag, $*$.

We have two classes of object types: open object types, $\{p_i : \tau_i|_{i=1}^{n}\}^\circ$, and closed object types, $\{p_i : \tau_i|_{i=1}^{n}\}^\bullet$. If an object has an open object type, it is referred to as an *open object*, and, if not, a *closed object*. Only open objects can be extended or shrank during execution, meaning that we can only add new fields or delete existing fields to/from open objects. The domain of a closed object is not allowed to change during execution and the types of its fields must remain the same. When an object is created, it is assumed to be open. After populating an object with all the fields that it should contain, the programmer must close it using the commit statement. Closing an object is essential if one intends to assign it to other variables, as our type system enforces that only closed objects can be referenced by more than one pointer. This behaviour is exemplified in the code snippet below:

```
1  x := {};
2  x.f:= 5 + 5;
3  commit(x);
4  y := x
```

By using the `commit` statement, `x` sees its type changing from $\{\mathtt{f} : number\}^\circ$ to $\{\mathtt{f} : number\}^\bullet$, making the object closed, and thus, unable to have its type further changed. If we were to remove the commit statement from line 3, we would render the assignment of `x` to `y` in line 4 illegal, as it would create a second reference to an open object.

2.2 Type System

Before proceeding to the definition of the type system, some preliminary definitions are required. In particular, we make use of *store typing environments* to associate each program variable with its corresponding type, and *global typing contexts* to associate each function identifier with the corresponding function type. The formal definitions are given below. We also introduce the notion of intersection of store typing environments in order to combine two store typing environments into a new one.

Definition 1 (Store Typing Environment)**.** *A store typing environment is a function* $\Gamma : Var \mapsto \mathrm{T}$ *mapping variables in* Var *to types in* T*.*

Definition 2 ($\Gamma \sqcap \Gamma'$)**.** *The intersection of two store typing environments,* Γ *and* Γ'*, denoted* $\Gamma \sqcap \Gamma' : Var \mapsto \mathrm{T}$*, is defined as:*

$$\Gamma \sqcap \Gamma'(x) = \begin{cases} \Gamma(x), & \textit{if } \Gamma(x) = \Gamma'(x) \\ \bot, & \textit{otherwise} \end{cases}$$

Definition 3 (Global Typing Context)**.** *A global typing context is a partial function* $\Delta : F \rightharpoonup \mathrm{T}$ *mapping function identifiers in the set of all function identifiers* F *to function types in* T*.*

$$\textsc{Variable}\ \frac{\tau = \Gamma(x)}{\Gamma \vdash x : \tau} \qquad \textsc{Value}\ \frac{Type(v) = \tau}{\Gamma \vdash v : \tau} \qquad \textsc{Unary Operation}\ \frac{\Gamma \vdash e : \tau_e \qquad \oplus(\tau_e) = \tau}{\Gamma \vdash \oplus(e) : \tau}$$

$$\textsc{Binary Operation}\ \frac{\Gamma \vdash e_1 : \tau_{e_1} \qquad \Gamma \vdash e_2 : \tau_{e_2} \qquad \otimes(\tau_{e_1}, \tau_{e_2}) = \tau}{\Gamma \vdash \otimes(e_1, e_2) : \tau}$$

Figure 2.2: Typing Rules for Expressions: $\Gamma \vdash e : \tau$

Typing Rules for Expressions Given a store typing environment Γ, an expression e and a type τ it is said that Γ types the expression e with type τ, written $\Gamma \vdash e : \tau$ as long as there is a derivation for it according to the rules defined in Figure 2.2. The rules there presented are explained below.

[VARIABLE] A store typing environment Γ types a variable x with type τ when the image of x in Γ is τ.

[VALUE] A store typing environment Γ types a value v with type τ when the type of v is in fact τ.

[UNARY OPERATION] A store typing environment Γ types a unary operation $\oplus(\tau_e) = \tau$ with type τ when Γ types e with type τ_e and the operation $\oplus$ over a type τ_e leads to a type τ.

[BINARY OPERATION] A store typing environment Γ types a binary operation $\otimes(\tau_{e_1}, \tau_{e_2})$ with type τ when Γ types e_1 with type τ_{e_1} and e_2 with type τ_{e_2}, and the operation $\otimes$ over types τ_{e_1} and τ_{e_2} leads to a type τ.

Typing Rules for Statements Given a function identifier g, a global typing context Δ, two store typing environment Γ_1 and Γ_2, and a statement s, the typing judgement $g, \Delta \vdash \{\Gamma_1\}\ s\ \{\Gamma_2\}$ means that s occurs within the body of g and that under the global typing context Δ, the execution of s on a variable store satisfying the initial store typing environment Γ_1 results in a variable store satisfying the final variable typing environment Γ_2.

The key insight of our type system is that we have to control aliasing. In particular, our type system enforces a no aliasing policy for open objects, which guarantees that open objects can only be accessed through a single program variable at a time. To ensure this, we do not allow open objects to be assigned to program variables and/or object fields. Once an object is closed, such assignments are allowed. We call this policy *no aliasing for open objects (NAOO)* and will discuss it thoroughly in the subsequent chapters of this thesis. In the rules, we make use of a predicate $\mathsf{Closed}(\tau)$ to determine whether or not the given type τ is closed. Primitive types are treated as closed object types, meaning that $\mathsf{Closed}(\tau)$ also holds when τ is a primitive type.

We are now at the position of describing the typing rules of our type system. These rules are given in Figure 2.3 and explained below.

[SKIP] Skip is always typable under all contexts and does not change the store typing environment.

[ASSIGNMENT] The rule first types the expression e being assigned, obtaining the type τ_e. It then checks that τ_e is closed in order to enforce the NAOO policy. Finally, it updates the type of x in the store typing environment to τ_e.

Skip

$$g, \Delta \vdash \{\Gamma\}\ \text{skip}\ \{\Gamma\}$$

Var Assignment

$$\frac{\Gamma \vdash e : \tau_e \qquad \text{Closed}(\tau_e)}{g, \Delta \vdash \{\Gamma\}\ x := e\ \{\Gamma[x \mapsto \tau_e]\}}$$

Field Lookup

$$\frac{\Gamma \vdash e : \{f_i : \tau_i|_{i=1}^{k}, f : \tau\}^{*} \qquad \text{Closed}(\tau)}{g, \Delta \vdash \{\Gamma\}\ x := e.f\ \{\Gamma[x \mapsto \tau]\}}$$

New Object

$$g, \Delta \vdash \{\Gamma\}\ x := \{\}\ \{\Gamma[x \mapsto \{\}^{\circ}]\}$$

Field Assignment - Open Exist

$$\frac{\Gamma \vdash e : \tau_e \qquad \Gamma(x) = \{f_i : \tau_i|_{i=1}^{k}\}^{\circ} \qquad \exists_{j \in \{1,\ldots,k\}} f = f_j \qquad \text{Closed}(\tau_e)}{g, \Delta \vdash \{\Gamma\}\ x.f := e\ \{\Gamma[x \mapsto \{f_i : \tau_i|_{i=1, i \neq j}^{k}, f : \tau_e\}^{\circ}]\}}$$

Field Assignment - Open Non Exist

$$\frac{\Gamma \vdash e : \tau_e \qquad \Gamma(x) = \{f_i : \tau_i|_{i=1}^{k}\}^{\circ} \qquad \forall_{i \in \{1,\ldots,k\}} f \neq f_i \qquad \text{Closed}(\tau_e)}{g, \Delta \vdash \{\Gamma\}\ x.f := e\ \{\Gamma[x \mapsto \{f_i : \tau_i|_{i=1}^{k}, f : \tau_e\}^{\circ}]\}}$$

Field Assignment - Close

$$\frac{\Gamma \vdash e : \tau_e \qquad \Gamma(x) = \{\ldots, f : \tau_f, \ldots\}^{\bullet} \qquad \tau_e = \tau_f \qquad \text{Closed}(\tau_e)}{g, \Delta \vdash \{\Gamma\}\ x.f := e\ \{\Gamma\}}$$

Field Delete

$$\frac{\Gamma(x) = \{f_i : \tau_i|_{i=1}^{k}\}^{\circ} \qquad \exists_{j \in \{1,\ldots,k\}} f = f_j}{g, \Delta \vdash \{\Gamma\}\ \text{delete}(x.f)\ \{\Gamma[x \mapsto \{f_i : \tau_i|_{i=1, i \neq j}^{k}\}^{\circ}]\}}$$

Commit

$$\frac{\Gamma(x) = \{f_i : \tau_i|_{i=1}^{k}\}^{\circ}}{g, \Delta \vdash \{\Gamma\}\ \text{commit}(x)\ \{\Gamma[x \mapsto \{f_i : \tau_i|_{i=1}^{k}\}^{\bullet}]\}}$$

If

$$\frac{\Gamma \vdash e : bool \qquad g, \Delta \vdash \{\Gamma_0\}\ s_1\ \{\Gamma_1\} \qquad g, \Delta \vdash \{\Gamma_0\}\ s_2\ \{\Gamma_2\}}{g, \Delta \vdash \{\Gamma_0\}\ \text{if}(e)\{s_1\}\ \text{else}\ \{s_2\}\ \{\Gamma_1 \sqcap \Gamma_2\}}$$

While

$$\frac{\Gamma \vdash e : bool \qquad g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma\}}{g, \Delta \vdash \{\Gamma\}\ \text{while}(e)\{s\}\ \{\Gamma\}}$$

Sequencing

$$\frac{g, \Delta \vdash \{\Gamma_0\}\ s_1\ \{\Gamma_1\} \qquad g, \Delta \vdash \{\Gamma_1\}\ s_1\ \{\Gamma_2\}}{g, \Delta \vdash \{\Gamma_0\}\ s_1; s_2\ \{\Gamma_2\}}$$

Function Call

$$\frac{\Gamma \vdash e_i : \tau_i|_{i=1}^{n} \qquad \Delta(f) = (\tau_1, \ldots, \tau_n) \mapsto \tau \qquad \text{Closed}(\tau_i)|_{i=1}^{n}}{g, \Delta \vdash \{\Gamma\}\ x := f(e|_{i=1}^{n})\ \{\Gamma[x \mapsto \tau]\}}$$

Return

$$\frac{\Delta(g) = (\tau_1, \ldots, \tau_n) \mapsto \tau \qquad \Gamma \vdash e : \tau}{g, \Delta \vdash \{\Gamma\}\ \text{return}(e)\ \{\Gamma\}}$$

Figure 2.3: Typing Rules for Statements: $f, \Delta \vdash \{\Gamma_1\}\ s\ \{\Gamma_2\}$

[ASSIGNMENT FROM FIELD] This rule is applied when assigning the value of an object field to a program variable. The rule first types the expression e that evaluates to the object being inspected and obtains the type τ of its field f. Then, it checks that τ is closed in order to maintain the NAOO invariant. Finally, the type of x is updated to τ in the store typing environment.

[NEW OBJECT] The new object statement is typable under all contexts, changing the type of x to $\{\}^{\circ}$.

[FIELD ASSIGNMENT OPEN EXIST] This rule is applied when assigning a closed value to an already existent field of an open object. The rule first types the expression e being assigned, obtaining the type τ_e, and checks if the latter is closed. Then, it checks if the object to which e is being assigned is open and the field to be assigned, f, is already contained in the object's fields, $f_i = f$ for some i. If so, the type of x is updated in the store typing environment so that the field f is mapped to τ_e.

[FIELD ASSIGNMENT OPEN NON EXIST] This rule is applied when assigning a closed value to a new field of an open object. The rule first types the expression e being assigned, obtaining the type τ_e, and checks if the latter is closed. Then, it checks if the object to which e is being assigned is open and the field to be assigned, f, is different from the existing fields. If so, the type of x is updated in the store typing environment so that it additionally maps the new field f to type τ_e.

[FIELD ASSIGNMENT CLOSE] This rule is applied when assigning a closed value to a field of a closed object. The rule first types the expression e being assigned, obtaining the type τ_e, and checks if the latter is closed. Then, it checks if the object to which e is assigned is also closed and obtains the type τ_f corresponding to the field being assigned. Finally, it checks if τ_e coincides with τ_f, leaving the store typing environment unchanged.

[FIELD DELETE] This rule is applied when deleting an existing field from an open object. The rule first checks if the programming variable x binds an open object and if that object has the field f. Finally, it updates the type of x in the store typing environment by removing the deleted field f.

[COMMIT] This rule is applied when closing an open object. The rule checks if the programming variable x binds an open object and, if so, it closes the type of x.

[IF] This rule is applied to an if statement. The rule first checks that the expression e guarding the if has type boolean. Then, it types both the then- branch and the else branch of the if statement, obtaining two final typing environments Γ_1 and Γ_2. Finally, resulting store typing environment is set to be the intersection of Γ_1 and Γ_2.

[WHILE] This rule is applied to a while statement. The rule first checks that the expression e guarding the while has type boolean. Then, it types the body of the while, checking that the resulting store typing environment precisely coincides with the starting one and leaving the store typing environment unchanged.

[SEQUENCING] This rule is applied to a sequencing statement $s_1; s_2$. The rule first types s_1 obtaining a new store typing environment Γ_1 and then types s_2 starting from Γ_1 and obtaining a new store typing environment Γ_2.

[FUNCTION CALL] This rule is applied to a function call statements. The rule first types the arguments of the function call, $e_i|_{i=1}^{n}$, additionally checking that they are all closed. Then, it sets the type of x in the store typing environment to the return type of the function.

[RETURN] This rule is applied to return statements. The rule types the expression being returned and checks if the obtained type coincides with the return type of the function g that contains the current return statement.

Typing functions A function $function\ f(x_1 : \tau_1, ..., x_n : \tau_n)\{s\}$ is said to be *typable* under the global typing context Δ, written $\Delta \vdash function\ f(x_1 : \tau_1, ..., x_n : \tau_n)\{s\}$, if its body is typable with respect to the typing environment obtained by mapping its formal parameters to their respective types. This concept is formally defined below.

$$\frac{\Gamma = [x_1 \mapsto \tau_1, ..., x_n \mapsto \tau_n] \qquad \Delta(f) = (\tau_1, ..., \tau_n) \rightarrow \tau \qquad f, \Delta \vdash \{\Gamma\} s \{\Gamma'\}}{\Delta \vdash function\ f(x_1 : \tau_1, ..., x_n : \tau_n)\{s\}}$$

A program p is said to be *typable* under a global typing context Δ, written $\Delta \vdash p$, if all functions in the range of the program are typable with respect to Δ. The notation $\Delta_r(f)$ is used to refer to the return type of f.

Typable and Non-Typable Examples Let us consider the two following code snippets, respectively showing a typable and a non-typable program. As the left program is typable, we can find a type derivation that witnesses its typability.

```
1   x := {};
2   x.f:= 5 + 5;
3   commit(x);
4   x.f:=2; // Typable
```

```
1   x := {};
2   x.f:= 5 + 5;
3   commit(x);
4   x.f:="test"; // Non-typable
```

```
1   x := {};
2   x.f:= 5 + 5;
3   x.f:="test";
4   commit(x);// Typable
```

To show that the left-hand side program is typable we start by applying the *sequencing* rule. After this, we apply the *new object* rule which has no associated constraints and updates the store typing environment, mapping the type of x to $\{\}^\circ$. Below, we present the type derivation for this program, denoting the statement presented in line j as sj and omitting, for the moment, the sub-tree corresponding to the statement $s2; s3; s4$, which is denoted by A.

$$\frac{\dfrac{}{g, \Delta \vdash \{\}\ x := \{\}\ \{[x, \{\}^\circ]\}} \qquad \boxed{A}}{g, \Delta \vdash \{\}\ s1; s2; s3; s4\ \{[x : \{f : number\}^\bullet]\}}$$

Let us now consider the type derivation A. The *sequencing* rule is again applied and is followed by the *field assignment open non exist* rule. This last rule can be applied as our expression $5 + 5$ has type $number$ which is a primitive type and therefore closed, x is an open object as checked by the store typing environment and f is not yet a field of x, thus the type of x is updated in our store typing environment to $\{f : number\}^{\circ}$. Below, we present the type derivation for this program, omitting for the moment, the sub-tree corresponding to the statement $s3; s4$, which is denoted by B.

$$\boxed{\textsf{A}}\ \frac{\dfrac{}{g, \Delta \vdash \{[x : \{\}^{\circ}]\}\ x.f := 5 + 5\ \{[x, \{f : number\}^{\circ}]\}} \qquad \boxed{\textsf{B}}}{g, \Delta \vdash \{[x : \{\}^{\circ}]\}\ s2; s3; s4\ \{[x : \{f : number\}^{\bullet}]\}}$$

Finally, let us consider the type derivation B. Once again the *sequencing* rule is applied. Given that x is an open object, the $commit$ rule can be applied and it becomes closed, which means that its type is updated to $\{f : number\}^{\bullet}$ in the store typing environment. Lastly, we apply the *field assignment close* rule, given that 2 and $x.f$ have the same closed type, $number$, and x binds a closed object. The derivation B is shown below in which we use num as an equivalent to the type $number$.

$$\frac{\dfrac{}{g, \Delta \vdash \{[x : \{f : num\}^{\circ}]\}\ \mathsf{commit}(x)\ \{[x : \{f : num\}^{\bullet}]\}} \qquad \dfrac{}{g, \Delta \vdash \{[x : \{f : num\}^{\bullet}]\}\ x.f := 2\ \{[x : \{f : num\}^{\bullet}]\}}}{g, \Delta \vdash \{[x : \{\}^{\circ}]\}\ s3; s4\ \{[x : \{f : number\}^{\bullet}]\}}$$

In contrast, our second program is not typable as it will reach the last step with the store typing environment as $(x, \{f : number\}^{\bullet})$ and there is no typing rule that enables a str type to be assigned to the field f of the closed object. As for our third program, as x is not yet closed, it does not require that $5 + 5$ and $"test"$ to be of the same type, thus it is typable.

Chapter 3

Big-Step Soundness

In this chapter we prove the soundness of our type system with respect to a big-step semantics of ECMA-SL. In Section 3.1, we start by defining state satisfiability, a relation that captures what it means for an ECMA-SL state to satisfy a given typing environment. Then, in Section 3.2, we introduce our big-step semantics for ECMA-SL. The main proof of soundness is finally given in Section 3.3, based on a number of semantic properties introduced and proven in this same section.

In order to simplify the exposition, the semantics of ECMA-SL that we first present does not model wrong executions and function calls. Sections 3.4 and 3.5 extend it to cater for these two aspects. Importantly, by modelling wrong executions explicitly, we are able to prove an important property of our type system: *fault avoidance* – that means that the execution of a well-typed program cannot generate an error.

3.1 ECMA-SL State Properties

ECMA-SL States An ECMA-SL state is composed of a heap $h : Loc \times Str \rightharpoonup \mathcal{V}$, mapping pairs of locations and string to values, and a store $\rho : Var \rightharpoonup \mathcal{V}$, mapping program variables to values.

Following well-established approaches for modelling the semantics of JavaScript [14, 15], instead of modelling a heap as a function from locations to objects, objects are not explicitly represented in the formalism. At the semantic level, an object can be seen as a *region of the heap*. More concretely, the object pointed to by location l corresponds to the set of cells whose first element is l. In the following, we write $h(l)$ to mean $\{(l, f) \mid (l, f) \in dom(h)\}$ and $dom(h(l))$ to mean $\{f \mid (l, f) \in dom(h)\}$.

3.1.1 State Satisfiability

In this subsection we define what it means for an ECMA-SL state to satisfy a given typing environment. To this end, we first extend the notion of typing environment to heaps, introducing the concept of *heap typing environment* and then give the formal definition of state satisfiability.

Heap Typing Environment In order to define state satisfiability we first introduce the concept of *heap typing environment* which maps each heap location to the type of the object it refers to.

Definition 4 (Heap Typing Environment)**.** *A heap typing environment is a partial function* $\Sigma : Loc \rightharpoonup \mathrm{T}$ *that maps locations from the set of heap locations* Loc *to the set of types* T*.*

In the following, we use $\Sigma(l, f)$ to refer to the type of the field f in the object pointed to by location l. Put formally: $\Sigma(l, f) = \tau_f \iff \exists\, \tau.\, \Sigma(l) = \tau \ \wedge\ \tau = \{..., f : \tau_f, ...\}^*$.

State Satisfiability We define the state satisfiability relation with the help of three auxiliary satisfiability relations:

- *Value Satisfiability:* describing what it means for a value v to satisfy a given type τ under a heap typing environment Σ – written $v \vDash_\Sigma \tau$;
- *Store Satisfiability:* describing what it means for a store ρ to satisfy a given store typing environment Γ under a heap typing environment Σ – written $\rho \vDash_\Sigma \Gamma$;
- *Heap Satisfiability:* describing what it means for a heap to satisfy a heap typing environment – written $h \vDash \Sigma$.

To avoid clutter, we use the notation $h, \rho \vDash \Sigma, \Gamma$ to mean that $h \vDash \Sigma$ *and* $\rho \vDash_\Sigma \Gamma$.

Definition 5 (Value Satisfiability)**.** *A value* v *is said to satisfy a type* τ *with respect to a heap typing environment* Σ*, written* $v \vDash_\Sigma \tau$*, if either:*

- v *is a number and* τ *is the number type;*
- v *is a string and* τ *is the string type;*
- v *is* $true$ *or* $false$ *and* τ *is the boolean type;*
- *if* v *is a location* l *and* $\Sigma(l) = \tau$

For primitive values, i.e. numbers, strings and booleans, the value satisfiability relation simply checks if the given type τ coincides with the type of the given value v, ignoring the supplied heap typing environment; for instance, it holds that $2 \vDash_\Sigma number$ and it does not hold that $2 \vDash_\Sigma string$. When it comes to an object location l, value satisfiability simply requires that the supplied type τ coincides with $\Sigma(l)$. Note that value satisfiability does not make sure that the structure of the object pointed to by a location l conforms with the type that is assigned to that location; that is the job of the definition of heap satisfiability, which connects the heap typing environment Σ with the actual heap h. Below, we give some additional examples that illustrate how the value satisfiability relation works.

Number Satisfiability	**Location Satisfiability**	**Location Non-Satisfiability**
Any Σ	$\Sigma(l) = \{age : number\}^\bullet$	$\Sigma(l) = \{age : number\}^\circ$
$6 \vDash_\Sigma number$	$l \vDash_\Sigma \{age : number\}^\bullet$	$\neg(l \vDash_\Sigma \{age : number\}^\bullet)$

Definition 6 (Store Satisfiability). *Given a heap typing environment Σ, a store ρ is said to satisfy a store typing environment Γ, written $\rho \models_\Sigma \Gamma$, if and only if:*

$$\forall_{x \in dom(\Gamma)} \rho(x) \models_\Sigma \Gamma(x)$$

Store satisfiability simply requires that the domain of the store typing environment Γ coincides with the domain of the given store ρ and that all values in the range of ρ satisfy their types given by Γ with respect to Σ. Notice that store satisfiability only requires the heap typing environment Σ for handling object locations.

Definition 7 (Heap Satisfiability). *A heap h is said to satisfy a heap typing environment Σ, written $h \models \Sigma$, if and only if:*

- $dom(h) = dom(\Sigma)$
- $\forall_{l \in dom(h)} dom(h(l)) = dom(\Sigma(l))$
- $\forall_{l \in dom(h)} \forall_{f \in dom(h(l))} h(l, f) \models_\Sigma \Sigma(l, f)$

Heap satisfiability requires that: **(1)** the domain of the heap coincides with the domain of the heap typing environment; **(2)** for each location in the heap, the fields stored in that location coincide with the fields contained in the corresponding object type (for instance, if location l contains the fields f_1 and f_2, its type must also declare the fields f_1 and f_2); **(3)** the value of each field of each object contained in the heap satisfies its corresponding type given by Σ.

3.1.2 No-aliasing Invariant

As stated before, in order to deal with aliasing and mutation, our type system enforces a simple invariant: *only closed objects can be referenced by more than one pointer*. We formalise this invariant as the state property given in the definition below.

Definition 8 (No Aliasing for Open Objects). *A heap h, a store ρ and a heap typing environment Σ are said to satisfy the n̲o a̲liasing for o̲pen o̲bjects (NAOO) property, written $NAOO(h, \rho, \Sigma)$, if and only if:*

- $\forall_{l \in dom(\Sigma)} \ \lfloor \Sigma(l) \rfloor = \circ \Rightarrow \neg \exists_{(l', f)} : \ h(l', f) = l$ $\qquad NAOO_1$
- $\forall_{l \in dom(\Sigma)} \ \lfloor \Sigma(l) \rfloor = \circ \Rightarrow \neg \exists_{x_1, x_2} \ x_1 \neq x_2 \wedge \rho(x_1) = \rho(x_2) = l$ $\qquad NAOO_2$

Essentially the n̲o a̲liasing for o̲pen o̲bjects (NAOO) property states that an open object can only be referenced by a single program variable; this means that: **(1)** it cannot be referenced by an object field ($NAOO_1$) and **(2)** it cannot be referenced by two distinct program variables x_1 and x_2 ($NAOO_2$).

Essentially, our type system enforces that objects can only be mutated if they are open, meaning that there is a single reference pointing to them. This guarantees that object mutation does not cause the type of a given reference (variable or object field) to become inconsistent with the type of its corresponding value. To better understand why this is the case, let us consider the object diagram given in Figure 3.1.

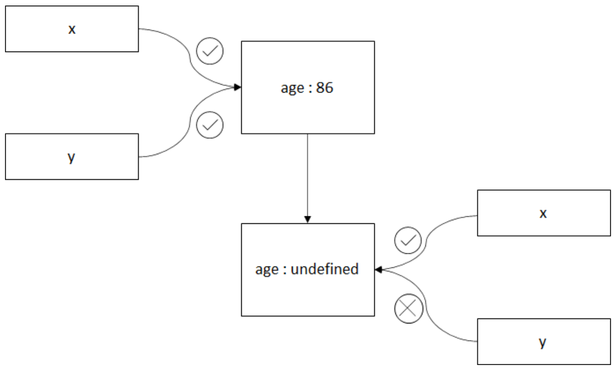

Figure 3.1: Difficulty posed by aliasing

Initially, we have two variables x and y pointing to an object with a single field age with value 86 of type number. Therefore, if not for the distinction between open and closed types, both x and y would have type $\{age : number\}$. Suppose now that we set the value of age to $undefined$ via the assignment $x.age := undefined$. In this case, for the type system to be sound, both the types of x and y must be set to $\{age : undefined\}$. However, that would require keeping track of all aliases of every program variable and object field at every program point, which is not tractable in practice. Instead, we only allow an object to be mutated if we can prove that there is a single reference pointing to it, thereby avoiding this type of inconsistency.

3.2 Big-Step Semantics

In this section we define a big-step semantics for ECMA-SL statements, ignoring for now function calls and erroneous executions. Our semantics for statements makes use of a simple big-step semantics for ECMA-SL expressions given in the figure below, where we use the notation $[\![e]\!]_\rho$ to mean the evaluation of the expression e in the store ρ. Note that, given that ECMA-SL expressions do not interact with the object heap, the semantics of expressions only depends on the variable store.

Variable	Value	Unary Operation	Binary Operation
$[\![x]\!]_\rho \triangleq \rho(x)$	$[\![v]\!]_\rho \triangleq v$	$[\![\oplus(e)]\!]_\rho \triangleq \underline{\oplus}([\![e]\!]_\rho)$	$[\![\otimes(e_1, e_2)]\!]_\rho \triangleq \underline{\otimes}([\![e_1]\!]_\rho, [\![e_2]\!]_\rho)$

Figure 3.2: Big-step semantics for expressions $[\![e]\!]_\rho \triangleq v$

Expression evaluation is straightforward:

- a variable x evaluates to the value to which it is mapped by the variable store;
- a value v evaluates to itself;
- to evaluate a unary operator expression $\oplus(e)$, the semantics first evaluates the argument expression e and then applies the semantic function corresponding to the given unary operator to the obtained value;

Skip
$$\langle \Sigma, h, \rho, \mathsf{skip}\rangle \Downarrow_i \langle \Sigma, h, \rho\rangle$$

Var Assignment
$$\frac{v = [\![e]\!]_\rho \qquad \mathsf{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x := e\rangle \Downarrow_i \langle \Sigma, h, \rho[x \mapsto v]\rangle}$$

Field Lookup
$$\frac{[\![e]\!]_\rho = l \qquad h(l, f) = v \qquad \mathsf{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x := e.f\rangle \Downarrow_i \langle \Sigma, h, \rho[x \mapsto v]\rangle}$$

New Object
$$\frac{l \notin dom(h) \qquad \Sigma' = \Sigma[l \mapsto \{\}^\circ] \qquad h' = h[l \mapsto \{\}]}{\langle \Sigma, h, \rho, x := \{\}\rangle \Downarrow_i \langle \Sigma', h', \rho[x \mapsto l]\rangle}$$

Field Assignment - Open
$$\frac{[\![e]\!]_\rho = v \qquad [\![x]\!]_\rho = l \qquad \tau = Type_\Sigma(v) \qquad \lfloor \Sigma(l) \rfloor = \circ \qquad \Sigma' = \Sigma[l \mapsto \Sigma(l)[f \mapsto \tau]] \qquad \mathsf{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x.f := e\rangle \Downarrow_i \langle \Sigma', h[(l, f) \mapsto v], \rho\rangle}$$

Field Assignment - Close
$$\frac{[\![e]\!]_\rho = v \qquad [\![x]\!]_\rho = l \qquad \Sigma(l, f) = Type_\Sigma(v) \qquad \lfloor \Sigma(l) \rfloor = \bullet \qquad (l, f) \in dom(h) \qquad \mathsf{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x.f := e\rangle \Downarrow_i \langle \Sigma, h[(l, f) \mapsto v], \rho\rangle}$$

Field Delete
$$\frac{[\![x]\!]_\rho = l \qquad \lfloor \Sigma(l) \rfloor = \circ}{\langle \Sigma, h, \rho, \mathsf{delete}(x.f)\rangle \Downarrow_i \langle \Sigma \backslash (l, f), h \backslash (l, f), \rho\rangle}$$

Commit
$$\frac{[\![x]\!]_\rho = l \qquad \lfloor \Sigma(l) \rfloor = \circ}{\langle \Sigma, h, \rho, \mathsf{commit}(x)\rangle \Downarrow_i \langle \Sigma[l \mapsto \Sigma(l)^\bullet], h, \rho\rangle}$$

If-True
$$\frac{[\![e]\!]_\rho = true \qquad \langle \Sigma, h, \rho, s_1\rangle \Downarrow_i \langle \Sigma', h', \rho'\rangle}{\langle \Sigma, h, \rho, \mathsf{if}(e)\{s_1\}\ \mathsf{else}\ \{s_2\}\rangle \Downarrow_i \langle \Sigma', h', \rho'\rangle}$$

If-False
$$\frac{[\![e]\!]_\rho = false \qquad \langle \Sigma, h, \rho, s_2\rangle \Downarrow_i \langle \Sigma', h', \rho'\rangle}{\langle \Sigma, h, \rho, \mathsf{if}(e)\{s_1\}\ \mathsf{else}\ \{s_2\}\rangle \Downarrow_i \langle \Sigma', h', \rho'\rangle}$$

While
$$\frac{\langle \Sigma, h, \rho, \mathsf{if}(e)\{s; \mathsf{while}(e)\{s\}\}\ \mathsf{else}\ \{\mathsf{skip}\}\rangle \Downarrow_i \langle \Sigma', h', \rho'\rangle}{\langle \Sigma, h, \rho, \mathsf{while}(e)\{s\}\rangle \Downarrow_i \langle \Sigma', h', \rho'\rangle}$$

Sequencing
$$\frac{\langle \Sigma, h, \rho, s_1\rangle \Downarrow_i \langle \Sigma_1, h_1, \rho_1\rangle \qquad \langle \Sigma_1, h_1, \rho_1, s_2\rangle \Downarrow_i \langle \Sigma_2, h_2, \rho_2\rangle}{\langle \Sigma, h, \rho, s_1; s_2\rangle \Downarrow_i \langle \Sigma_2, h_2, \rho_2\rangle}$$

Figure 3.3: Big-step semantics for statements: $\langle \Sigma, h, \rho, s\rangle \Downarrow_i \langle \Sigma', h', \rho'\rangle$

- to evaluate a binary operator expression $\otimes(e_1, e_2)$, the semantics first evaluates the argument expressions e_1 and e_2 and then applies the semantic function corresponding to the given binary operator to the obtained values.

We are now at the position to define our semantic judgement for statements, which has the form $\langle \Sigma, h, \rho, s\rangle \Downarrow_i \langle \Sigma', h', \rho'\rangle$, meaning that the evaluation of the statement s in the heap h and store ρ results in the heap h' and store ρ'. In order to reason about the types of the objects in the heap, we have to instrument the semantics to keep track of the types of the objects created at runtime. To this end, the semantic judgement for statements additionally include the initial and final heap typing environments, respectively Σ and Σ'. The semantic rules are given in Figure 3.3.

The proposed semantics enforces the *no aliasing for open objects* (NAOO) invariant. To this end, before every assignment, the semantics checks if the value being assigned is either a primitive value or a closed object; only in such cases is the assignment allowed to go through. To avoid clutter, we introduce the predicate $\mathsf{Closed}_\Sigma(v)$ to mean that v is either a primitive value or a closed object.

Definition 9 (Closed Values)**.** *Let Σ be a heap typing environment, a value v is said to be of a closed with respect to Σ, written $\mathsf{Closed}_{\Sigma}(v)$, if and only if it is of a primitive type or if $\lfloor \Sigma(v) \rfloor = \bullet$.*

The semantic rules are explained below:

[SKIP] The skip transition is always possible and leaves the heap typing environment, heap and store unchanged.

[VAR ASSIGNMENT] This rule is used when a closed value is being assigned to a program variable. The rule first evaluates the expression e obtaining the value v, which is then assigned to the variable x in the store ρ.

[FIELD LOOKUP] This rule is applied when assigning a closed value from an object field to a program variable. The rule first evaluates the expression e obtaining the object location l. Then, it obtains the value v associated with the field f in the object pointed to by l and updates the value of the variable x to v in the store ρ.

[NEW OBJECT] This rule is applied when assigning a new object to a variable x. The rule finds a location l which is not in the domain of the heap and adds such location to both heap and heap typing environment, mapping it to an empty object and to an empty open object type, respectively. The rule also maps x to l in the store.

[FIELD ASSIGNMENT - OPEN] This rule is applied when assigning a closed value to a field of an open object. The rule first evaluates the expression e obtaining the value v, and the location l of the program variable x. Then, it updates the field f being modified and its type in the heap and heap typing environment, respectively.

[FIELD ASSIGNMENT - CLOSE] This rule is applied when assigning a closed value to a field of a close object. The rule first evaluates the expression e obtaining the value v, and the location l of the program variable x. Then, it verifies if the already existing field f of l has the same type as v and, if so, it updates the value of the field f to v in the heap h.

[FIELD DELETE] This rule is applied when deleting a field from an open object. The rule first obtains the location l of the program variable x. Then it removes the pair (l, f) from the domain of both heap and heap typing environment.

[COMMIT] This rule is applied when closing an open object. The rule first obtains the location l of the program variable x. Afterwards it updates the type associated with the location l to a closed type in the store typing environment Σ.

[IF-TRUE] This rule is applied when evaluating an if statement and its associated boolean expression evaluates to $true$. The rule delegates the work to another transition which has as statement the then branch. The final heap typing environment, heap and store are the ones obtained through the support transition.

[IF-FALSE] This rule is applied when evaluating an if statement and its associated boolean expression evaluates to $false$. The rule delegates the work to another transition which has as statement the else branch. The final heap typing environment, heap and store are the ones obtained through the support transition.

[WHILE] This rule is applied when evaluating a $while$ statement. The rule simply delegates the work to another transition which has a conditional statement with the original condition. The then branch is the sequencing of the body of the while followed by the $while$ statement itself and the else branch is simply the $skip$ statement. The resulting heap typing environment, heap and store are the same as the ones obtained through this supporting transition.

[SEQUENCING] The rule is used to evaluate a sequence of two statements $s_1; s_2$. The rule first evaluates the statement s_1, obtaining the heap typing environment Σ_1, heap h_1, and store ρ_1. The rule next evaluates the statement s_2 on Σ_1, h_1 and ρ_1, obtaining the heap typing environment Σ_2, heap h_2, and store ρ_2.

3.3 Soundness - Type Safety

In this section we prove the first soundness property of the proposed type system: type safety. We prove that our type system satisfies this property with respect to the operational semantics defined in Section 3.2. Essentially, we say that our type system is *safe* in that the execution of a typed statement preserves state satisfiability. More formally, if a statement s is typable with respect to a given function g, typing context Δ and store typing environments Γ and Γ', i.e. $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$, and if one executes s in a state (h, ρ) such that $h, \rho \vDash \Sigma, \Gamma$, for a given heap typing environment Σ, then if the execution terminates, it will do so in a state satisfying the final heap and store typing environments; put formally:

$$(g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\} \ \wedge\ h, \rho \vDash \Sigma, \Gamma \ \wedge\ \langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho' \rangle) \ \Rightarrow\ h', \rho' \vDash \Sigma', \Gamma' \tag{3.1}$$

In order to establish the safety of our type system, we make use of a number of auxiliary properties regarding: **(1)** preservation of the NAOO invariant; **(2)** soundness of expression typing; and **(3)** satisfiability preservation for heap and store updates. The following subsections expand on these three classes of properties and the section concludes with the main soundness proof.

3.3.1 Preservation of the NAOO Invariant

The proposed type system enforces the NAOO invariant, which states that only closed objects can be referenced by more than one pointer at a time. However, we do not prove that the type system does enforce the NAOO invariant directly. Instead, we instrumented the operational semantics so that it also enforces the NAOO invariant and will later prove that typable programs cannot be rejected by the semantics for violating the NAOO invariant. Lemma 1 proves that the operational semantics preservers the NAOO invariant.

Lemma 1 (NAOO Properties)**.** *Let Σ and Σ' be heap typing environments, h and h' heaps, ρ and ρ' stores, and s a statement such that $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho' \rangle$ and $NAOO(h, \rho, \Sigma)$; then it holds that*

$NAOO(h', \rho', \Sigma')$.

Proof. Both $NAOO$ properties have to be considered, therefore we segment our proof in two parts:

NAOO-I Suppose that for some location l, $\exists_{(l',f)}:\ h(l', f) = l$. Then, for this to happen, there had to be a step $s = x.f := e$ such that $[\![e]\!]_\rho = l$ and $[\![x]\!]_\rho = l'$. Therefore the transition rule "field assignment - open" had to be applied, thus $\lfloor\Sigma(l)\rfloor = \bullet$. Moreover by rule inspection, l can not be open after being closed. Thus, since $\exists_{(l',f)}:\ h(l', f) = l \Rightarrow \lfloor\Sigma(l)\rfloor \neq \circ$. Therefore we conclude that $\lfloor\Sigma(l)\rfloor = \circ \Rightarrow \neg\exists_{(l',f)}:\ h(l', f) = l$.

NAOO-II Suppose that for some x_1, x_2 such that $x_1 \neq x_2$ and $l \in dom(\Sigma)$ we have that $\rho(x_1) = \rho(x_2) = l$. Then, for this to happen one of the following steps had to happen for $i = 1$ and for $i = 2$:

- $s_1 = x_i := e$ where $[\![e]\!]_\rho = l$
- $s_2 = x_i := e.f$ where $[\![e]\!]_\rho = k$ and $k.f = l$ (for some location k)
- $s_3 = x_i := \{\}$ where $[\![x_i]\!]_{\rho'} = l$

Notice that if s_1 or s_2 is applied, then $\lfloor\Sigma(l)\rfloor = \bullet$. It is also impossible to apply s_3 for both x_1 and x_2, since the corresponding semantic rule requires that $l \notin dom(h)$. Moreover, and also by rule inspection, l can not be open after being closed. Thus, since $\exists_{x_1,x_2}\ x_1 \neq x_2 \wedge \rho(x_1) = \rho(x_2) = l \Rightarrow \neg\lfloor\Sigma(l)\rfloor = \circ$, we conclude that $\lfloor\Sigma(l)\rfloor = \circ \Rightarrow \neg\exists_{x_1,x_2}\ x_1 \neq x_2 \wedge \rho(x_1) = \rho(x_2) = l$ ☐

3.3.2 Well-Typed Expressions

Our type system for statements relies on a simple type system for expressions. Unsurprisingly, in order to establish the safety of the type system for statements, we first have to establish the safety of our type system for expressions. To establish this, we have to prove that if an expression e is given type τ_e by the type system in a store typing environment Γ and if the evaluation of e in a store ρ satisfying Γ yields a value v; then, the value v must be of type τ_e. Lemma 2 formally establishes this property.

Lemma 2 (Well-typed Expressions - Safety). *Let e be an expression, τ_e a type, ρ a store, Σ a heap typing environment and Γ store typing environment. Suppose that $\Gamma \vdash e : \tau_e$, $\rho \vDash_\Sigma \Gamma$ and $[\![e]\!]_\rho = v$. Then $v \vDash_\Sigma \tau_e$.*

Proof. Assume that $\Gamma \vdash e : \tau_e\ (hyp.1)$, $\rho \vDash_\Sigma \Gamma\ (hyp.2)$ and $[\![e]\!]_\rho = v\ (hyp.3)$. Therefore we have that:

Case e is a variable x (hyp.4)

- $\Gamma(x) = \tau_e$ (1) - hyp.1 + hyp.4
- $\rho(x) \vDash_\Sigma \Gamma(x)$ (2) - hyp.2
- $\rho(x) = v$ (3) - hyp.3 + hyp.4
- $v \vDash_\Sigma \tau_e$ (4) - (1) + (2) + (3)

Case e is value v' (hyp.4)

- $\tau_e = Type(v')$ (1) - hyp.1 + hyp.4
- $v' = v$ (2) - hyp.3 + hyp.4

- $\tau_e = Type(v)$ (3) - (1) + (2)
- $v \vDash_\Sigma \tau_e$ (4) - (3)

Case e is $\oplus(e')$ (hyp.4)

- $\exists \tau'_e : \Gamma \vdash e' : \tau'_e$ (1) - hyp.1 + hyp.4
- $\exists v' : [\![e']\!]_\rho = v'$ (2) - hyp.3 + hyp.4
- $v' \vDash_\Sigma \tau'_e$ (3) - (1) + (2) + hyp.2 + hyp.ind.
- $+(v') \vDash_\Sigma \underline{\oplus}(\tau'_e)$ (4) - (3)
- $v \vDash_\Sigma \tau_e$ (5) - (1) + (2) + (4) + hyp.4

Case e is $\otimes(e_1, e_2)$ (hyp.4)

- $\exists \tau_{e_1}, \tau_{e_2} : \Gamma \vdash e_1 : \tau_{e_1}$ and $\Gamma \vdash e_2 : \tau_{e_2}$ (1) - hyp.1 + hyp.4
- $\exists v_1, v_2 : [\![e_1]\!]_\rho = v_1$ and $[\![e_2]\!]_\rho = v_2$ (2) - hyp.3 + hyp.4
- $v_1 \vDash_\Sigma \tau_{e_1}$, $v_2 \vDash_\Sigma \tau_{e_2}$ (3) - (1) + (2) + hyp.2 + hyp.ind.
- $\underline{\otimes}(v_1, v_2) \vDash_\Sigma \underline{\otimes}(\tau_{e_1}, \tau_{e_2})$ (4) - (3)
- $v \vDash_\Sigma \tau_e$ (5) - (1) + (2) + (4) + hyp.4

$\square$

3.3.3 Satisfiability Preservation

In the following we present a number of lemmas that characterize the conditions under which updates to the store, heap, store typing environment, and heap typing environment preserve the satisfiability relation introduced in Section 3.1. For clarity, we group these lemmas into four groups depending on the context in which the lemma is used: field update, field deletion, object creation, and object closing.

Field Update Regarding the update of object fields, we start with a simple lemma stating that if a heap h satisfies a heap typing environment Σ, and one updates the field (l, f) by a value of the exact same type of its previous value, then the updated heap continues to satisfy the original heap typing environment.

Lemma 3 (Heap Update - Type Unchanged). *Let h be a heap, l a location, f a field, τ a type and Σ a heap typing environment. Suppose that $h(l, f) \vDash_\Sigma \tau$, $v' \vDash_\Sigma \tau$, $h' = h[(l, f) \mapsto v']$ and $h \vDash \Sigma$. Then $h' \vDash \Sigma$.*

A more general version of this lemma requires additional conditions: the object being updated has to be open and the $NAOO_1$ property must be satisfied. This generalized version, Lemma 4, states that if the heap h satisfies the heap typing environment Σ and the field (l, f) is updated with a closed value v in h and with type τ in Σ, then the updated heap also satisfies the updated heap typing environment.

Lemma 4 (Heap Update). *Let h and h' be heaps, Σ and Σ' heap typing environments, v a value, l a location and τ a type. Suppose that $h \vDash \Sigma$, $h' = h[(l, f) \mapsto v]$, $\Sigma' = \Sigma[l \mapsto \Sigma(l)[f \mapsto \tau]]$, $NAOO_1(h, \Sigma)$, $\mathsf{Closed}_\Sigma(v)$, $\lfloor \Sigma(l) \rfloor = \circ$ and $v \vDash_\Sigma \tau$. Then $h' \vDash \Sigma'$.*

The preservation of store satisfaction is more involved. Suppose we are given a store ρ that satisfies a store typing environment Γ with respect to a heap typing environment Σ. Then, suppose that a variable x in the domain of ρ is bound to the location l pointing to an open object with a field f. Finally, suppose that we update the field f of the object pointed to by l with a closed value v and also update Γ and Σ accordingly (both $\Gamma(x)$ and $\Sigma(l)$ must have the type of f updated to the type of v). In this case, the original store continues to satisfy the updated store typing environment.

Lemma 5 (Store Update). *Let Σ and Σ' be heap typing environments, Γ and Γ' store typing environments, ρ a store, x a variable, f a field and τ a type. Suppose that $\rho \vDash_\Sigma \Gamma$, $\Sigma' = \Sigma[\rho(x) \mapsto \Sigma(\rho(x))[f \mapsto \tau]]$, $\Gamma' = \Gamma[x \mapsto \Gamma(x)[f \mapsto \tau]]$, $\lfloor \Sigma(\rho(x)) \rfloor = \circ$ and $NAOO_2(\rho, \Sigma)$. Then $\rho \vDash_{\Sigma'} \Gamma'$.*

Field Deletion Regarding field deletion there are two lemmas to consider: heap delete and store delete. The Heap Delete Lemma states that when a heap h satisfies a heap typing environment Σ, and a field (l, f) is removed from both h and Σ, the resulting heap continues to satisfy the resulting heap typing environment, as long as the location l points to an open object with respect to the updated heap typing environment.

Lemma 6 (Heap Delete). *Let h and h' be heaps, Σ and Σ' heap typing environments, ρ a store, l a location and f a field. Suppose that $h \vDash \Sigma$, $h' = h \backslash (l, f)$, $\Sigma' = \Sigma \backslash (l, f)$ and $NAOO_1(h, \Sigma)$, $\lfloor \Sigma(l) \rfloor = \circ$. Then $h' \vDash \Sigma'$.*

The Store Delete Lemma is a bit more involved. Suppose we are given a store ρ that satisfies a store typing environment Γ with respect to a heap typing environment Σ. Then, suppose that a variable x in the domain of ρ is bound to the location l pointing to an open object with a field f. Finally, suppose that we delete the field f of the object pointed to by l and update Γ and Σ accordingly (both $\Gamma(x)$ and $\Sigma(l)$ must be updated so that f is removed from their sets of fields). In this case, the original store continues to satisfy the updated store typing environment with respect to the updated heap typing environment.

Lemma 7 (Store Delete). *Let Σ and Σ' be heap typing environments, Γ and Γ' store typing environments, ρ a store, x a variable and f a field. Suppose that $\rho \vDash_\Sigma \Gamma$, $\Sigma' = \Sigma \backslash (\rho(x), f)$, $\Gamma' = \Gamma \backslash (x, f)$, $\lfloor \Sigma(\rho(x)) \rfloor = \circ$ and $NAOO_2(\rho, \Sigma)$. Then $\rho \vDash_{\Sigma'} \Gamma'$.*

Object Creation Of all the operations listed in this section, object creation is the only one that does not require the $NAOO$ properties. This stems from the fact that all the other operations deal with objects that already existed before, whereas this one deals with the creation of a new object. The Heap New Lemma states that if a heap h satisfies a heap typing environment Σ, and one creates a new object and extends Σ with a mapping from l to an empty open object type, then the resulting heap still satisfies the resulting heap typing environment.

Lemma 8 (Heap New). *Let h and h' be heaps, Σ and Σ' heap typing environments and l a location. Suppose that $h \vDash \Sigma$, $h' = h[l \mapsto \{\}]$, $\Sigma' = \Sigma[l \mapsto \{\}^\circ]$ and $l \notin dom(h)$. Then $h' \vDash \Sigma'$.*

Let us now consider the Store New Lemma. Suppose we are given a store ρ that satisfies a store typing environment Γ with respect to a heap typing environment Σ. Then, suppose that one creates a new object and assigns it to a variable x, updating Γ and Σ accordingly; both Γ and Σ must be updated in order to respectively maps x and the newly created location to the empty object type. In this case, the updated store satisfies the updated store typing environment with respect to the updated heap typing environment.

Lemma 9 (Store New)**.** *Let Σ and Σ' be heap typing environments, Γ and Γ' store typing environments, ρ and ρ' stores and x a variable. Suppose that $\rho \vDash_{\Sigma} \Gamma$, $\rho' = \rho[x \mapsto l]$, $\Sigma' = \Sigma[\rho'(x) \mapsto \{\}^{\circ}]$, $\Gamma' = \Gamma[x \mapsto \{\}^{\circ}]$ and $l \notin dom(\Sigma)$. Then $\rho' \vDash_{\Sigma'} \Gamma'$.*

Object Closing Lastly, we consider the preservation properties associated with the $commit$ command. The Heap Close Lemma states that if a heap h satisfies a heap typing environment Σ and one closes the type of an open object in h by setting its openness flag to $\bullet$ in Σ, then the resulting heap typing environment is still satisfied by h.

Lemma 10 (Heap Close)**.** *Let h ba a heap, Σ and Σ' heap typing environments, ρ a store and l a location. Suppose that $h \vDash \Sigma$, $\Sigma' = \Sigma[l \mapsto \Sigma(l)^{\bullet}]$, $NAOO_1(h, \Sigma)$ and $\lfloor \Sigma(l) \rfloor = \circ$. Then $h \vDash \Sigma'$.*

The Store Close Lemma states that when a store ρ satisfies a store typing environment Γ with respect to a heap typing environment Σ, and there is a variable in the store which its image is of an open type, then we might close it in both Γ and Σ and ρ still satisfies the resulting store typing environment with respect to the resulting heap typing environment.

Lemma 11 (Store Close)**.** *Let Σ and Σ' be heap typing environments, Γ and Γ' store typing environments, ρ a store and x a variable. Suppose that $\rho \vDash_{\Sigma} \Gamma$, $\Sigma' = \Sigma[\rho(x) \mapsto \Sigma(\rho(x))^{\bullet}]$, $\Gamma' = \Gamma[x \mapsto \Gamma(x)^{\bullet}]$, $\lfloor \Sigma(\rho(x)) \rfloor = \circ$ and $NAOO_2(\rho, \Sigma)$. Then $\rho \vDash_{\Sigma'} \Gamma'$.*

3.3.4 Soundness - Type Safety

Theorem 1 states that the proposed type system satisfies the Type Safety property. This theorem makes use the various lemmas introduced in the previous sections as well as other less important properties given in the Appendix. Figure 3.4 illustrates the dependencies between the discussed lemmas and the main theorem.

Theorem 1 (Soundness - Type Safety)**.** *Let g be a function and Δ a typing context. Let h be a heap, ρ a store and s a statement. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$, $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho' \rangle$ and $h, \rho \vDash \Sigma, \Gamma$. Then $h', \rho' \vDash \Sigma', \Gamma'$.*

Proof. The proof follows by induction on the derivation of $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho' \rangle$. Hence, assuming that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$ (hyp.1), $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho' \rangle$ (hyp.2) and $h, \rho \vDash \Sigma, \Gamma$ (hyp.3).

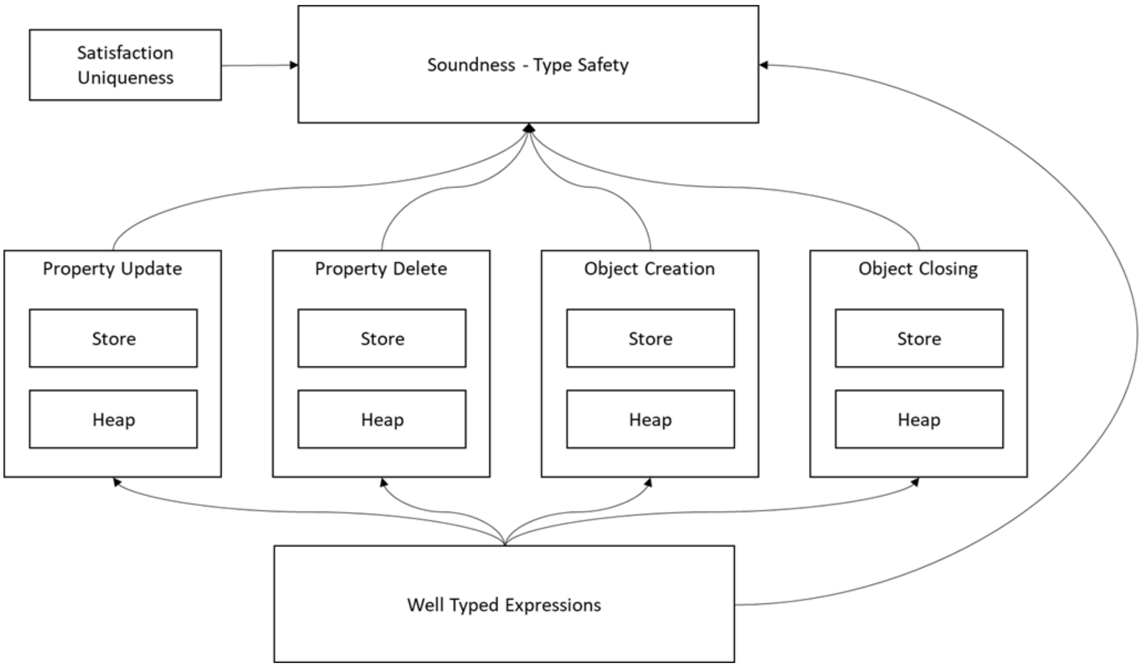

Figure 3.4: Relation between lemmas and theorems for the proof of Type Safety

Base Cases:

[SKIP] Suppose that $s = \mathsf{skip}$ (hyp.4). We have that:

- $g, \Delta \vdash \{\Gamma\}\ \mathsf{skip}\ \{\Gamma\}$ is applied. (1) - hyp.4
- $\langle \Sigma, h, \rho, \mathsf{skip} \rangle \Downarrow_i \langle \Sigma, h, \rho \rangle$ (2) - hyp.4
- $\Gamma' = \Gamma$ (3) - hyp.1 + (1)
- $\Sigma' = \Sigma, h' = h, \rho' = \rho$ (4) - hyp.2 + (2)
- $h', \rho' \vDash \Sigma', \Gamma'$ (5) - (3) + (4) + hyp.3

[VAR ASSIGNMENT] Suppose that $x := e$ (hyp.4). We have that:

- $\frac{\Gamma \vdash e : \tau_e \ \ Closed(\tau_e)}{g, \Delta \vdash \{\Gamma\}\ x := e\ \{\Gamma[x \mapsto \tau_e]\}}$ is applied (1) - hyp.4
- $\Gamma \vdash e : \tau_e$ (2) - (1)
- $\langle \Sigma, h, \rho, x := e \rangle \Downarrow_i \langle \Sigma, h, \rho[x \mapsto \llbracket e \rrbracket_\rho] \rangle$ (3) - hyp.4
- $\Gamma' = \Gamma[x \mapsto \tau_e]$ (4) - hyp.1 + (1)
- $\Sigma' = \Sigma, h' = h, \rho' = \rho[x \mapsto \llbracket e \rrbracket_p]$ (5) - hyp.2 + (3)
- $\forall_{y \in dom(\rho), y \neq x}\ \Gamma(y) = \Gamma'(y)$ e $\rho(y) = \rho'(y)$ (6) - (4) + (5)
- Se $y = x$ então $\rho'(x) = \llbracket e \rrbracket_\rho$ e $\rho'(y) \vDash \Gamma'(y)$ (7) - (2) + (4) + (5) + Lemma WTE-Safety
- $\forall_{y \in dom(\Gamma')}\ \rho'(y) \vDash_\Sigma \Gamma'(y)$ (8) - (6) + (7)
- $h', \rho' \vDash \Sigma', \Gamma'$ (9) - (5) + (8) + hyp.3

[FIELD LOOKUP] Suppose that $x := e.f$(hyp.4). We have that:

- $\frac{\Gamma \vdash e:\{f_i:\tau_i|_{i=1}^{k}, f:\tau\}^* \ Closed(\tau)}{g,\Delta \vdash \{\Gamma\} x := e.f \{\Gamma[x \mapsto \tau]\}}$ is applied (1) - hyp.4
- $\Gamma \vdash e : \{f_i : \tau_i|_{i=1}^{k}, f : \tau\}^*$ (2) - (1)
- $\langle \Sigma, h, \rho, x := e\rangle \Downarrow_i \langle \Sigma, h, \rho[x \mapsto h([\![e]\!]_\rho, [\![f]\!]_\rho)]\rangle$ (3) - hyp.4
- $\Gamma' = \Gamma[x \mapsto \tau]$ (4) - hyp.1 + (1)
- $\Sigma' = \Sigma, h' = h, \rho' = \rho[x \mapsto h([\![e]\!]_\rho, [\![f]\!]_\rho)]$ (5) - hyp.2 + (3)
- $\forall_{y \in dom(\rho), y \neq x}\ \Gamma(y) = \Gamma'(y)$ e $\rho(y) = \rho'(y)$ (6) - (4) + (5)
- Se $y = x$ então $\rho'(x) = h([\![e]\!]_\rho, f)$ e $\rho'(y) \models \Gamma'(y)$ (7) - (2) + (4) + (5) + Lemma WTE-Safety
- $\forall_{y \in dom(\Gamma')}\ \rho'(y) \models_\Sigma \Gamma'(y)$ (8) - (6) + (7)
- $h', \rho' \models \Sigma', \Gamma'$ (9) - (5) + (8) + hyp.3

[FIELD ASSIGNMENT - OPEN] Suppose that $s = x.f := e$ (hyp.4) and $\lfloor \Sigma([\![x]\!]_\rho) \rfloor = \circ$ (hyp.5).

- $\langle \Sigma, h, \rho, x.f := e\rangle \Downarrow_i \langle \Sigma', h[(l, f) \mapsto v], \rho\rangle$ (1) - hyp.2 + hyp.4 + hyp.5
- $[\![x]\!]_\rho = l$ (2) - (1)
- $[\![e]\!]_\rho = v$ (3) - (1)
- $\tau = Type_\Sigma(v)$ (4) - (1)
- $\Sigma' = \Sigma[l \mapsto \Sigma(l)[f \mapsto \tau]], h' = h[(l, f) \mapsto v], \rho' = \rho$ (5) - (1)
- Closed$_\Sigma(v)$ (6) - (1)
- $l \models_\Sigma \Gamma(x)$ (7) - hyp.3 + (2) + Lemma WTE-Safety
- $\lfloor \Gamma(x) \rfloor = \circ$ (8) - hyp.5 + (7)

We have two cases to consider:

Case $(l, f) \notin dom(h)$ (hyp.6):

- $f \notin dom(\Sigma(l))$ (9.1.1) - hyp.3 + hyp.6
- $f \notin dom(\Gamma(x))$ (9.1.2) - (9.1.1)
- $\frac{\Gamma \vdash e:\tau_e \ \Gamma(x)=\{f_i:\tau_i|_{i=1}^{k}\}^\circ, \forall_{i \in \{1,\ldots,k\}} f \neq f_i}{g,\Delta \vdash \{\Gamma\} x.f := e \{\Gamma[x \mapsto \{f_i:\tau_i|_{i=1}^{k}, f:\tau_e\}^\circ]\}}$ is applied. (9.1.3) - (8) + (9.1.2) + hyp.1
- $\Gamma \vdash e : \tau_e$ (9.1.4) - (9.1.3)
- $\Gamma(x) = \{f_i : \tau_i|_{i=1}^{k}\}^\circ$ (9.1.5) - (9.1.3)
- $\Gamma' = \Gamma[x \mapsto \{f_i : \tau_i|_{i=1}^{k}, f : \tau_e\}^\circ]$ (9.1.6) - (9.1.3)
- $v \models_\Sigma \tau_e$ (9.1.7) - (3) + (9.1.4) + hyp.3 + Lemma WTE-Safety
- $\tau_e = \tau$ (9.1.9) - (9.1.7) + (4) + Lemma Satisfaction Uniqueness
- $h' \models \Sigma'$ (9.1.10) - (5) + (6) + (9.1.8) + hyp.3 + hyp.5 + $NAOO_1$ + Lemma Heap Update
- $\Gamma' = \Gamma[x \mapsto \Gamma(x)[f \mapsto \tau]]$ (9.1.11) - (9.1.5) + (9.1.6) + (9.1.9)
- $\rho' \models_{\Sigma'} \Gamma'$ (9.1.12) - (5) + (6) + (9.1.11) + hyp.3 + hyp.5 + $NAOO_2$ + Lemma Store Update
- $h', \rho' \models \Sigma', \Gamma'$ (9.1.13) - (9.1.10) + (9.1.12)

Case $(l, f) \in dom(h)$ (hyp.6):

- $f \in dom(\Sigma(l))$ (9.2.1) - hyp.3 + hyp.6

- $f \in dom(\Gamma(l))$ (9.2.2) - (9.2.1)
- $\frac{\Gamma \vdash e:\tau_e \;\; \Gamma(x)=\{f_i:\tau_i|_{i=1}^{k}\}^{\circ}, \exists_{j\in\{1,\ldots,k\}} f=f_j}{g,\Delta \vdash \{\Gamma\} x.f:=e \{\Gamma[x \mapsto \{f_i:\tau_i|_{i=1,i\neq j}^{k}, f:\tau_e\}^{\circ}]\}}$ is applied. (9.1.3) - (8) + (9.1.2) + hyp.1
- $\Gamma \vdash e : \tau_e$ (9.1.4) - (9.1.3)
- $\Gamma(x) = \{f_i : \tau_i|_{i=1}^{k}\}^{\circ}$ (9.1.5) - (9.1.3)
- $\Gamma' = \Gamma[x \mapsto \{f_i : \tau_i|_{i=1,i\neq j}^{k}, f : \tau_e\}^{\circ}]$ (9.1.6) - (9.1.3)
- $v \vDash_{\Sigma} \tau_e$ (9.1.7) - (3) + (9.1.4) + hyp.3 + Lemma WTE-Safety
- $v \vDash_{\Sigma} \tau$ (9.1.8) - (4)
- $\tau_e = \tau$ (9.1.9) - (9.1.7) + (9.1.8) + Lemma Satisfaction Uniqueness
- $h' \vDash \Sigma'$ (9.1.10) - (5) + (6) + (9.1.8) + hyp.3 + hyp.5 + $NAOO_1$ + Lemma Heap Update
- $\Gamma' = \Gamma[x \mapsto \Gamma(x)[f \mapsto \tau]]$ (9.1.11) - (9.1.5) + (9.1.6) + (9.1.9)
- $\rho' \vDash_{\Sigma'} \Gamma'$ (9.1.12) - (5) + (6) + (9.1.11) + hyp.3 + hyp.5 + $NAOO_2$ + Lemma Store Update
- $h', \rho' \vDash \Sigma', \Gamma'$ (9.1.13) - (9.1.10) + (9.1.12)

[FIELD ASSIGNMENT - CLOSE] Suppose that $s = x.f := e$ (hyp.4) and $\lfloor \Sigma(\llbracket x \rrbracket_{\rho}) \rfloor = \bullet$ (hyp.5). We have that:

- $\frac{\Gamma \vdash e:\tau_e \;\; \Gamma(x)=\{\ldots,f:\tau_f,\ldots\}^{\bullet} \;\; \tau_e=\tau_f}{g,\Delta \vdash \{\Gamma\} x.f:=e \{\Gamma\}}$ is applied (1) - hyp.1 + hyp.4 + hyp.5
- $\Gamma \vdash e : \tau_e$ (2) - (1)
- $\Gamma(x) = \{..., f : \tau_f, ...\}^{\bullet}$ (3) - (1)
- $\tau_e = \tau_f$ (4) - (1)
- $\langle \Sigma, h, \rho, x.f := e\rangle \Downarrow_i \langle \Sigma, h[(l, f) \mapsto v], \rho\rangle$ (5) - hyp.2 + hyp.4 + hyp.5
- $\llbracket e \rrbracket_{\rho} = v$ (6) - (5)
- $\llbracket x \rrbracket_{\rho} = l$ (7) - (5)
- $\Sigma(l, f) = Type_{\Sigma}(v)$ (8) - (5)
- $h(l, f) = v_f$ (9) - (5)
- $\Gamma' = \Gamma$ (10) - (1)
- $\Sigma' = \Sigma, h' = h[(l, f) \mapsto v], \rho' = \rho$ (11) - (5)
- $\rho' \vDash_{\Sigma'} \Gamma'$ (12) - (10) + (11) + hyp.3

We now have two situations to consider to show that $\forall_{l \in dom(\Sigma)} h'(l) \vDash \Sigma'(l)$:

- If $\hat{l} \neq l$ then $h'(\hat{l}) = h(\hat{l})$ therefore $h'(\hat{l}) \vDash \Sigma'(\hat{l})$ (13.1) - (11)
- If $\hat{l} = l$ we have that: (13.2)
  - $\forall_{g \in dom(h) \setminus f} : h'(\hat{l}, g) = h(\hat{l}, g)$ (13.2.2) - (11)
  - $\forall_{g \in dom(h) \setminus f} : h'(\hat{l}, g) \vDash_{\Sigma'} \Sigma'(\hat{l}, g)$ (13.2.3) - (11) + (13.2.2)
  - $v \vDash_{\Sigma} \tau_e$ (13.2.4) - (2) + (6)
  - $v \vDash_{\Sigma'} \tau_e$ (13.2.5) - (8) + (13.2.4)
  - $v \vDash_{\Sigma'} \tau_f$ (13.2.6) - (4) + (13.2.5) + (8)
  - $h'(\hat{l}, f) \vDash_{\Sigma'} \Sigma'(\hat{l}, f)$ (13.2.7) - (3) + (11) + (13.2.6)

  - $h'(\hat{l}) \vDash \Sigma'(\hat{l})$ (13.2.8) - (13.2.1) + (13.2.3) + (13.2.7)

Therefore:

- $h' \vDash \Sigma'$ (14) - (13.1) + (13.2)
- $h', \rho' \vDash \Sigma', \Gamma'$ (15) - (12) + (14)

[NEW OBJECT] Suppose that $s = x := \{\}$(hyp.4). We have that:

- $g, \Delta \vdash \{\Gamma\} x := \{\} \{\Gamma[x \mapsto \{\}^\circ]\}$ is applied. (1) - hyp.1 + hyp.4
- $\langle \Sigma, h, \rho, x := \{\}\rangle \Downarrow_i \langle \Sigma', h', \rho[x \mapsto l]\rangle$ (2) - hyp.2 + hyp.4
- $l \notin dom(h)$ (3) - (2)
- $\Gamma' = \Gamma[x \mapsto \{\}^\circ]$ (4) - (1)
- $\Sigma' = \Sigma[l \mapsto \{\}^\circ], h' = h[l \mapsto \{\}], \rho' = \rho[x \mapsto l]$ (5) - (2)
- $h' \vDash \Sigma'$ (6) - (3) + (4) + (5) + hyp.3 + Lemma Geap New
- $\rho' \vDash_{\Sigma'} \Gamma'$ (7) - (3) + (5) + hyp.3 + Lemma Store New
- $h', \rho' \vDash \Sigma', \Gamma'$ (8) - (6) + (7)

[COMMIT] Suppose that $s = commit(x)$(hyp.4). We have that:

- $\frac{\lfloor \Gamma(x) \rfloor = \circ}{g, \Delta \vdash \{\Gamma\}\ commit(x)\ \{\Gamma[x \mapsto \Gamma(x)^\bullet]\}}$ is applied. (1) - hyp.1 + hyp.4
- $\langle \Sigma, h, \rho, commit(x)\rangle \Downarrow_i \langle \Sigma[l \mapsto \Sigma(l)^\bullet], h, \rho\rangle$ (2) - hyp.2 + hyp.4
- $[\![x]\!]_\rho = l$ (3) - (2)
- $\lfloor \Sigma(l) \rfloor = \circ$ (4) - (2)
- $\Gamma' = \Gamma[x \mapsto \Gamma(l)^\bullet]$ (5) - (1)
- $\Sigma' = \Sigma[l \mapsto \Sigma(l)^\bullet], h' = h, \rho' = \rho$ (6) - (2)
- $h' \vDash \Sigma'$ (7) - (4) + (6) + hyp.3 + Lemma Heap Close
- $\rho' \vDash_{\Sigma'} \Gamma'$ (8) - (3) + (4) +(5) + (6) + hyp.3 + Lemma Store Close
- $h', \rho' \vDash \Sigma', \Gamma'$ (9) - (7) + (8)

[FIELD DELETE] Suppose that $s = \text{delete}(x)$(hyp.4). We have that:

- $\frac{\Gamma(x) = \{f_i : \tau_i |_{i=1}^{k}\}^\circ, \exists_{j \in \{1, \ldots, k\}} f = f_j}{g, \Delta \vdash \{\Gamma\} delete\ x.f \{\Gamma[x \mapsto \{f_i : \tau_i |_{i=1, i \neq j}^{k}\}^\circ]\}}$ is applied. (1) - hyp.1 + hyp.4
- $\langle \Sigma, h, \rho, \text{delete}(x.f)\rangle \Downarrow_i \langle \Sigma \backslash (l, f), h \backslash (l, f), \rho\rangle$ (2) - hyp.2 + hyp.4
- $[\![x]\!]_\rho = l$ (3) - (2)
- $\Gamma' = \Gamma \backslash (x, f)$ (4) - (2)
- $\Sigma' = \Sigma \backslash (l, f), h' = h \backslash (l, f), \rho' = \rho$ (5) - (1)
- $h' \vDash \Sigma'$ (6) - (4) + hyp.1 + Lemma Heap Delete
- $\rho' \vDash_{\Sigma'} \Gamma'$ (7) - (4) hyp.1 + Lemma Store Delete
- $h', \rho' \vDash \Sigma', \Gamma'$

Induction cases:

[IF-TRUE] It follows that $s = \mathsf{if}(e)\{s_1\}$ else $\{s_2\}$ (hyp.4) and $[\![e]\!]_\rho = true$ (hyp.5). We have that:

- $\frac{g,\Delta\vdash e:bool \;\; g,\Delta\vdash\{\Gamma_0\}s_1\{\Gamma_1\} \;\; g,\Delta\vdash\{\Gamma_0\}s_2\{\Gamma_2\}}{g,\Delta\vdash\{\Gamma_0\}if(e)\{s_1\}else\{s_2\}\{\Gamma_1\sqcap\Gamma_2\}}$ is applied. (1) - hyp.1 + hyp.4
- $g, \Delta \vdash \{\Gamma\}s_1\{\Gamma_1\}$ (2) - (1)
- $\langle\Sigma, h, \rho, \mathsf{if}(e)\{s_1\}$ else $\{s_2\}\rangle \Downarrow_i \langle\Sigma', h', \rho'\rangle$ (3) - hyp.2 + hyp.4
- $\langle\Sigma, h, \rho, s_1\rangle \Downarrow_i \langle\Sigma', h', \rho'\rangle$ (4) - (3) + hyp.5
- $h', \rho' \vDash \Sigma', \Gamma_1$ (5) - (2) + (4) + hyp.3 + hyp.ind.
- $h', \rho' \vDash \Sigma', \Gamma_1 \sqcap \Gamma_2$ (6) - (5) + Lemma Weakening
- $\Gamma' = \Gamma_1 \sqcap \Gamma_2$ (7) - (1)
- $h', \rho' \vDash \Sigma', \Gamma'$ (8) - (4) + (6) + (7)

[IF-FALSE] It follows that $s = \mathsf{if}(e)\{s_1\}$ else $\{s_2\}$ (hyp.4) and $[\![e]\!]_\rho = false$ (hyp.5). We have that:

- $\frac{g,\Delta\vdash e:bool \;\; g,\Delta\vdash\{\Gamma_0\}s_1\{\Gamma_1\} \;\; g,\Delta\vdash\{\Gamma_0\}s_2\{\Gamma_2\}}{g,\Delta\vdash\{\Gamma_0\}if(e)\{s_1\}else\{s_2\}\{\Gamma_1\sqcap\Gamma_2\}}$ is applied. (1) - hyp.1 + hyp.4
- $g, \Delta \vdash \{\Gamma\}s_1\{\Gamma_2\}$ (2) - (1)
- $\langle\Sigma, h, \rho, \mathsf{if}(false)\{s_1\}$ else $\{s_2\}\rangle \Downarrow_i \langle\Sigma', h', \rho'\rangle$ (3) - hyp.2 + hyp.4
- $\langle\Sigma, h, \rho, s_2\rangle \Downarrow_i \langle\Sigma', h', \rho'\rangle$ (4) - (3) + hyp.5
- $h', \rho' \vDash \Sigma', \Gamma_2$ (5) - (2) + (4) + hyp.3 + hyp.ind.
- $h', \rho' \vDash \Sigma', \Gamma_1 \sqcap \Gamma_2$ (6) - (5) + Lemma weakning
- $\Gamma' = \Gamma_1 \sqcap \Gamma_2$ (7) - (1)
- $h', \rho' \vDash \Sigma', \Gamma'$ (8) -(4) + (5) + (7)

[WHILE] Suppose that $s = \mathsf{while}(e)\{s\}$ (hyp.4). We have that:

- $\frac{g,\Delta\vdash e:bool \;\; g,\Delta\vdash\{\Gamma\}s\{\Gamma\}}{g,\Delta\vdash\{\Gamma\}while(e)\{s\}\{\Gamma\}}$ is applied. (1) - hyp.1 + hyp.4
- $g, \Delta \vdash \{\Gamma\}s\{\Gamma\}$ (2) - (1)
- $g, \Delta \vdash e : bool$ (3) - (1)
- By derivation, $g, \Delta \vdash \{\Gamma\}$ $\mathsf{if}(e)\{s; \mathsf{while}(e)\{s\}\}$ else $\{\mathsf{skip}\}$ $\{\Gamma\}$ (4) - (2) + (3) + hyp.1

$$\dfrac{g, \Delta \vdash e : bool \qquad \dfrac{g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma\} \qquad g, \Delta \vdash \{\Gamma\}\ \mathsf{while}(e)\{s\}\ \{\Gamma\}}{g, \Delta \vdash \{\Gamma\}\ s; \mathsf{while}(s)\{s\}\ \{\Gamma\}} \qquad g, \Delta \vdash \{\Gamma\}\ \mathsf{skip}\ \{\Gamma\}}{g, \Delta \vdash \{\Gamma\}\ \mathsf{if}(e)\{s; \mathsf{while}(e)\{s\}\}\ \mathsf{else}\ \{\mathsf{skip}\}\ \{\Gamma\}}$$

- $\langle\Sigma, h, \rho, \mathsf{while}(e)\{s\}\rangle \Downarrow_i \langle\Sigma', h', \rho'\rangle$ (5) - hyp.2 + hyp.4
- $\langle\Sigma, h, \rho, \mathsf{if}(e)\{s; \mathsf{while}(e)\{s\}\}$ else $\{\mathsf{skip}\}\rangle \Downarrow_i \langle\Sigma', h', \rho'\rangle$ (6) - (5)
- $h', \rho' \vDash \Sigma', \Gamma$ (7) - (4) + (6) + hyp.3 + hyp.ind.
- $\Gamma' = \Gamma$ (8) - (1)
- $h', \rho' \vDash \Sigma', \Gamma'$ (9) - (7) + (8)

[SEQUENCING] Suppose that $s = s_1; s_2$ (hyp.4). We have that:

- $\frac{g,\Delta\vdash\{\Gamma\}s_1\{\Gamma_1\}\ g,\Delta\vdash\{\Gamma_1\}s_2\{\Gamma_2\}}{g,\Delta\vdash\{\Gamma\}s_1;s_2\{\Gamma_2\}}$ is applied. (1) - hyp.1 + hyp.4
- $g, \Delta \vdash \{\Gamma\}s_1\{\Gamma_1\}$ (2) - (1)
- $g, \Delta \vdash \{\Gamma_1\}s_2\{\Gamma_2\}$ (3) - (1)
- $\langle\Sigma, h, \rho, s_1; s_2\rangle \Downarrow_i \langle\Sigma', h', \rho'\rangle$ (4) - hyp.2 + hyp.4
- $\langle\Sigma, h, \rho, s_1\rangle \Downarrow_i \langle\Sigma_1, h_1, \rho_1\rangle$ (5) - (4)
- $\langle\Sigma_1, h_1, \rho_1, s_2\rangle \Downarrow_i \langle\Sigma_2, h_2, \rho_2\rangle$ (6) - (4)
- $h_1, \rho_1 \vDash \Sigma_1, \Gamma_1$ (7) - (2) + (4) + hyp.3 + hyp.ind.
- $h_2, \rho_2 \vDash \Sigma_2, \Gamma_2$ (8) - (3) + (6) + (7) + hyp.ind.
- $\Gamma' = \Gamma_2$ (9) - (1)
- $\Sigma' = \Sigma_2, h' = h_2, \rho' = \rho_2$ (10) - (4)
- $h', \rho' \vDash \Sigma', \Gamma'$ (11) - (8) + (9) + (10)

Therefore $h', \rho' \vDash \Sigma', \Gamma'$

$\square$

3.4 Soundness - Fault Avoidance

In this section, we extend our operational semantics to take into account erroneous executions and prove the standard fault avoidance property of sound type systems: *well-typed programs cannot go wrong*. To this end, we extend the operational semantics defined in Section 3.2 with explicit error derivations, writing: $\langle\Sigma, h, \rho, s\rangle \Downarrow_i \lightning$ to mean that evaluation of the statement s in the heap h, store ρ, and heap typing environment Σ leads to an execution error. One can leverage these error derivations to formally define fault avoidance. Essentially, if a statement s is typable with respect to a given function g, typing context Δ and store typing environments Γ and Γ', i.e. $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$, and if one executes s in a state (h, ρ) such that $h, \rho \vDash \Sigma, \Gamma$, for a given heap typing environment Σ, then the execution of s will not result in a runtime error; put formally:

$$(g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}\ \wedge\ h, \rho \vDash \Sigma, \Gamma)\ \Rightarrow\ \langle\Sigma, h, \rho, s\rangle \not\Downarrow_i \lightning \tag{3.2}$$

3.4.1 Error Executions

Figure 3.5 extends the operational semantics given in Section 3.2 with a set of explicit error derivations. These derivations model the runtime errors that can occur during the execution of an ECMA-SL statement. We consider the following five types of runtime errors: **(1)** branching on a value that is not of boolean type; **(2)** creating a second reference to an open object, thereby violating the NAOO invariant; **(3)** updating a field of a closed object to a value of a different type; **(4)** adding a new field to a closed object; and **(5)** deleting a field from a closed object. Below, we explain some of the rules; the others are analogous.

OPEN RHS - ASSIGNMENT

$$\frac{v = [\![e]\!]_\rho \qquad \neg\mathsf{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x := e\rangle \Downarrow_i \lightning}$$

OPEN RHS - LOOKUP

$$\frac{[\![e]\!]_\rho = l \qquad h(l,f) = v \qquad \neg\mathsf{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x := e.f\rangle \Downarrow_i \lightning}$$

OPEN RHS - FIELD ASSIGNMENT

$$\frac{[\![e]\!]_\rho = v \qquad [\![x]\!]_\rho = l \qquad \neg\mathsf{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x.f := e\rangle \Downarrow_i \lightning}$$

CLOSED OBJECT - ILLEGAL UPDATE

$$\frac{[\![e]\!]_\rho = v \qquad [\![x]\!]_\rho = l \qquad \Sigma(l) = \{f_i : \tau_i|_{i=1}^{n}, f : \tau\}^\bullet \qquad Type_\Sigma(v) \neq \tau}{\langle \Sigma, h, \rho, x.f := e\rangle \Downarrow_i \lightning}$$

CLOSED OBJECT - ILLEGAL NEW FIELD

$$\frac{[\![x]\!]_\rho = l \qquad \Sigma(l) = \{f_i : \tau_i|_{i=1}^{n}\}^\bullet \qquad \forall_i\ f_i \neq f}{\langle \Sigma, h, \rho, x.f := e\rangle \Downarrow_i \lightning}$$

CLOSED OBJECT - ILLEGAL DELETION

$$\frac{[\![x]\!]_\rho = l \qquad \lfloor Sigma(l) \rfloor = \bullet}{\langle \Sigma, h, \rho, \mathsf{delete}(x.f)\rangle \Downarrow_i \lightning}$$

IF - ILLEGAL GUARD

$$\frac{[\![e]\!]_\rho = v \qquad Type_\Sigma(v) \neq bool}{\langle \Sigma, h, \rho, \mathsf{if}(e)\{s_1\}\ \mathsf{else}\ \{s_2\}\rangle \Downarrow_i \lightning}$$

IF-TRUE - ERROR

$$\frac{[\![e]\!]_\rho = true \qquad \langle \Sigma, h, \rho, s_1\rangle \Downarrow_i \lightning}{\langle \Sigma, h, \rho, \mathsf{if}(e)\{s_1\}\ \mathsf{else}\ \{s_2\}\rangle \Downarrow_i \lightning}$$

IF-FALSE - ERROR

$$\frac{[\![e]\!]_\rho = false \qquad \langle \Sigma, h, \rho, s_2\rangle \Downarrow_i \lightning}{\langle \Sigma, h, \rho, \mathsf{if}(e)\{s_1\}\ \mathsf{else}\ \{s_2\}\rangle \Downarrow_i \lightning}$$

WHILE - ILLEGAL GUARD

$$\frac{[\![e]\!]_\rho = v \qquad Type_\Sigma(v) \neq bool}{\langle \Sigma, h, \rho, \mathsf{while}(e)\{s\}\rangle \Downarrow_i \lightning}$$

WHILE - ERROR

$$\frac{\langle \Sigma, h, \rho, \mathsf{if}(e)\{s; \mathsf{while}(e)\{s\}\}\ \mathsf{else}\ \{\mathsf{skip}\}\rangle \Downarrow_i \lightning}{\langle \Sigma, h, \rho, \mathsf{while}(e)\{s\}\rangle \Downarrow_i \lightning}$$

SEQUENCING - ERROR 1

$$\frac{\langle \Sigma, h, \rho, s_1\rangle \Downarrow_i \lightning}{\langle \Sigma, h, \rho, s_1; s_2\rangle \Downarrow_i \lightning}$$

SEQUENCING - ERROR 2

$$\frac{\langle \Sigma, h, \rho, s_1\rangle \Downarrow_i \langle \Sigma_1, h_1, \rho_1\rangle \qquad \langle \Sigma, h, \rho, s_2\rangle \Downarrow_i \lightning}{\langle \Sigma_1, h_1, \rho_1, s_1; s_2\rangle \Downarrow_i \lightning}$$

Figure 3.5: Big-Step semantics for statements - erroneous executions: $\langle \Sigma, h, \rho, s\rangle \Downarrow_i \lightning$

[OPEN RHS - ASSIGNMENT] This rule is applied when an open object is assigned to a variable. The rule simply evaluates the expression e obtaining the value v which is open thus leading to an erroneous execution.

[OPEN RHS - LOOKUP] This rule is applied when an open field of an object is assigned to a variable. The rule first evaluates the expression e obtaining the value l. Then, it obtains the value v associated with the field f in the object pointed to by l which is open thus leading to an erroneous execution.

[OPEN RHS - FIELD ASSIGNMENT] This rule is applied when assigning an open value to an object's field. The rule first evaluates the expression e obtaining the value v, and the location l of the program variable x. Then, as v is open it leads to an erroneous execution.

[CLOSED OBJECT - ILLEGAL UPDATE] This rule is applied when assigning a value to a field of a closed object, whose type differs from that of the given value. The rule first evaluates the expression e obtaining the value v, and the variable x obtaining the location l. Then, the rule checks the type τ associated with the field f of l, which is different from the type of v, thus leading to an erroneous execution.

[CLOSED OBJECT - ILLEGAL NEW FIELD] This rule is applied when assigning a value to a non-existing field of a closed object. The rule first evaluates the variable x obtaining the location l. Then, as the field f is not present in l it leads to an erroneous execution.

[CLOSED OBJECT - ILLEGAL DELETION] This rule is applied when deleting the field of a closed object. The rule first evaluates the variable x obtaining the location l. Then as l points to a closed object the rule leads to an erroneous execution.

[IF - ILLEGAL GUARD] This rule is applied when evaluating an if statement whose condition does not evaluate to a boolean. The rule simply evaluates the expression e obtaining the value v which is not a boolean, thus leading to an erroneous execution.

[IF-TRUE - ERROR] This rule is applied when evaluating an if statement whose condition evaluates to $true$ and then branch leads to a erroneous execution. The rule simply evaluates the expression e obtaining the value $true$, and since the then branch leads to a erroneous execution so does this rule.

[IF-FALSE - ERROR] This rule is applied when evaluating an if statement whose condition evaluates to $false$ and else branch leads to a erroneous execution. The rule simply evaluates the expression e obtaining the value $false$, and since the then branch leads to a erroneous execution so does this rule.

[WHILE - ILLEGAL GUARD] This rule is applied when evaluating a $while$ statement whose condition does not evaluate to a boolean. The rule simply evaluates the expression e obtaining the value v which is not a boolean, thus leading to an erroneous execution.

[WHILE - ERROR] This rule is applied when evaluating a $while$ statement and its resolving leads to an erroneous execution. Since the resolving statement leads to an erroneous execution so does this statement.

[SEQUENCING - ERROR 1] This rule is applied when evaluating a $sequencing$ statement and its first step evaluates to a wrong execution. Since this step leads to an erroneous execution so does this statement.

[SEQUENCING - ERROR 2] This rule is applied when evaluating a $sequencing$ statement and its second step evaluates to an erroneous execution. The rule first verifies that the first step of the sequential statement is correctly evaluated and, since the second step evaluates to an erroneous execution so does this statement.

3.4.2 Soundness - Fault Avoidance

Theorem 2 states that the proposed type system satisfies the Fault Avoidance property.

Theorem 2 (Soundness - Fault Avoidance)**.** *Let g be a function and Δ a Typing Context. Let h be a heap, ρ a store and s a statement. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$ and $h, \rho \vDash \Sigma, \Gamma$ then it is not the case that $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \lightning$.*

Proof. Assuming that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$ (hyp.1) and $h, \rho \vDash \Sigma, \Gamma$ (hyp.2). Suppose also by contradiction that $\langle \Sigma, h, \rho, s \rangle \not\Downarrow_i \lightning$ (hyp.3).
Below, we present some of the cases; the others are analogous.
[VAR ASSIGNMENT] Suppose that $s = x := e$ (hyp.4). We have that:

- $\frac{\Gamma\vdash e:\tau_e \ \ Closed(\tau_e)}{g,\Delta\vdash\{\Gamma\}\ x:=e\ \{\Gamma[x\mapsto\tau_e]\}}$ is applied (1) - hyp.1 + hyp.4
- $\Gamma \vdash e : \tau_e$ (2) - (1)
- $Closed(\tau_e)$ (3) - (1)
- $\exists_v :\ [\![e]\!]_\rho = v$ (4) - hyp.3 + hyp.4
- $\neg\text{Closed}_\Sigma(v)$ (5) - hyp.3 + hyp.4
- $v \vDash_\Sigma \tau_e$ (6) - (2) + (4) + hyp.2 + Lemma WTE-Safety
- $\text{Closed}_\Sigma(v)$ (7) - (3) + (6) + Lemma Closed Values
- Contradiction (8) - (5) + (7)

[FIELD ASSIGNMENT - OPEN] Suppose that $s = x.f := e$ (hyp.4) and $\lfloor\Sigma([\![x]\!]_\rho)\rfloor = \circ$ (hyp.5). We have that:

- $\frac{\Gamma\vdash e:\tau_e \ \ \Gamma(x)=\{f_i:\tau_i|_{i=1}^{k}\}^\circ,\forall_{i\in\{1,\ldots,k\}} f\neq f_i \ \ \text{Closed}(\tau_e)}{g,\Delta\vdash\{\Gamma\}x.f:=e\{\Gamma[x\mapsto\{f_i:\tau_i|_{i=1}^{k},f:\tau_e\}^\circ]\}}$ or
  $\frac{\Gamma\vdash e:\tau_e \ \ \Gamma(x)=\{f_i:\tau_i|_{i=1}^{k}\}^\circ,\exists_{j\in\{1,\ldots,k\}} f=f_j \ \ \text{Closed}(\tau_e)}{g,\Delta\vdash\{\Gamma\}x.f:=e\{\Gamma[x\mapsto\{f_i:\tau_i|_{i=1,i\neq j}^{k},f:\tau_e\}^\circ]\}}$ is applied. (1) - hyp.1 + hyp.4 + hyp.5
- $\Gamma \vdash e : \tau_e$ (2) - (1)
- $Closed(\tau_e)$ (3) - (1)
- $\exists_v :\ [\![e]\!]_\rho = v$ (4) - hyp.1 + hyp.2
- $\neg\text{Closed}_\Sigma(v)$ (5) - (4) + hyp.3 + hyp.4
- $v \vDash_\Sigma \tau_e$ (6) - (2) + (4) + hyp.2 + Lemma WTE-Safety
- $\text{Closed}_\Sigma(v)$ (7) - (3) + (6) + Lemma Closed Values
- Contradiction (8) - (5) + (7)

[FIELD ASSIGNMENT - CLOSE] Suppose that $s = x.f := e$ (hyp.4) and $\lfloor\Sigma([\![x]\!]_\rho)\rfloor = \bullet$ (hyp.5). We have that:

- $\frac{\Gamma\vdash e:\tau_e \ \ \Gamma(x)=\{\ldots,f:\tau_f,\ldots\}^\bullet \ \ \tau_e=\tau_f}{g,\Delta\vdash\{\Gamma\}x.f:=e\{\Gamma\}}$ is applied (1) - hyp.1 + hyp.4 + hyp.5
- $\Gamma \vdash e : \tau_e$ (2) - (1)
- $\tau_e = \tau_f$ (3) - (1)
- $\exists_v :\ [\![e]\!]_\rho = v$ (4) - hyp.3 + hyp.4
- $\Sigma([\![x]\!]_\rho, f) \neq \Sigma(v) = \Sigma([\![e]\!]_\rho)$ (5) - (4) + hyp.3 + hyp.4
- $v \vDash_\Sigma \tau_e$ (6) - (2) + (4) + hyp.2 + Lemma WTE-Safety
- $\tau_f = \Gamma(x, f) = \Sigma([\![x]\!]_\rho, f)$ (7) - (1) + hyp.2 + hyp.4
- $\Sigma([\![e]\!]_\rho) = \Sigma(v) = \tau_e$ (8) - (5) + (6) + hyp.2
- $\tau_e = \tau_f$ (9) - (5) + (7) + (8)
- Contradiction (10) - (3) + (9)

[SEQUENCING] Suppose that $s = s_1; s_2$ (hyp.4). We have that:

- $\frac{g,\Delta\vdash\{\Gamma\}s_1\{\Gamma_1\} \ \ g,\Delta\vdash\{\Gamma_1\}s_2\{\Gamma_2\}}{g,\Delta\vdash\{\Gamma\}s_1;s_2\{\Gamma_2\}}$ is applied. (1) - hyp.1 + hyp.4
- $g, \Delta \vdash \{\Gamma\}\ s_1\ \{\Gamma_1\}$ (2) - (1)
- $g, \Delta \vdash \{\Gamma_1\}\ s_2\ \{\Gamma_2\}$ (3) - (1)

  Since $\langle\Sigma, h, \rho, s\rangle\overline{\Downarrow_i}\lightning$ (hyp.3), we have two cases to consider:

  Case $\langle\Sigma, h, \rho, s_1\rangle \Downarrow_i \lightning$ (hyp.4)

- We do not have $\langle \Sigma, h, \rho, s_1 \rangle \Downarrow_i \lightning$ (4.1.1) - (2) + hyp.2 + hyp.ind.
- Contradiction (4.1.2) - (4.1.1) + hyp.4

Case $\langle \Sigma, h, \rho, s_2 \rangle \Downarrow_i \lightning$ (hyp.4) and $\langle \Sigma, h, \rho, s_1 \rangle \Downarrow_i \langle \Sigma_1, h_1, \rho_1 \rangle$ (hyp.5)

- $\Sigma_1, \Gamma_1 \vDash h_1, \rho_1$ (4.2.1) - (2) + hyp.2 + hyp.5 + soundness
- We do not have $\langle \Sigma_1, h_1, \rho, s_2 \rangle \Downarrow_i \lightning$ (4.2.2) - (2) + (4.2.1) + hyp.ind.
- Contradiction (4.2.3) - (4.2.2) + hyp.4

□

3.5 Function and Return

In this section, we extend the operational semantics introduced in Section 3.2 with support for function calls and return statements. To this end, we have to change the format of the semantic judgment so that it additionally produces an outcome, which captures the flow of execution. Modified semantic judgements have the form: $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho', o \rangle$ signifying that the evaluation of the statement s in the heap h, store ρ, and heap typing environment Σ results in the heap h', store ρ', heap typing environment Σ', and outcome o. Outcomes are given by the following grammar:

$$o ::= \mathit{Cont} \mid \mathit{Err} \mid \mathit{Ret}(v) \tag{3.3}$$

We consider three types of outcomes: **(1)** the continuation outcome Cont, signifying that the execution may proceed with the next statement; **(2)** the error outcome Err, signifying that the execution generated an error and must therefore be terminated; and **(3)** the return outcome $\mathit{Ret}(v)$, signifying that the code of the function that is currently executing returned the value v.

Figure 3.6 gives a selection of the extended semantic rules. In particular, it includes the rules that handle function calls, the return statement, and sequencing, as well as a few other rules for illustrative purposes. Note that the rules introduced in Figures 3.3 and 3.5 can be straightforwardly adapted to this setting. Non-error derivations are simply extended with the continuation outcome Cont. For instance, the Skip rule becomes: $\langle \Sigma, h, \rho, \mathsf{skip} \rangle \Downarrow_i \langle \Sigma, h, \rho, \mathit{Cont} \rangle$. Error derivations are even simpler: instead of generating the special symbol $\lightning$, they are modified to generate the error outcome Err. For instance, the rule *open rhs - assignment* becomes:

$$\frac{v = [\![e]\!]_\rho \qquad \neg\mathsf{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x := e \rangle \Downarrow_i \langle \Sigma, h, \rho, \mathit{Err} \rangle}$$

In the following, we explain the rules given in Figure 3.6:

[FIELD DELETE] This rule is applied when deleting a field from an object. The rule first evaluates the variable x to the location l whose field f is to be deleted and checks that it points to an open object. Afterwards, it removes the pair consisting of the location and field, (l, f) from both the domain of the heap

Field Delete
$$\frac{[\![x]\!]_\rho = l \qquad \lfloor \Sigma(l) \rfloor = \circ}{\langle \Sigma, h, \rho, \mathsf{delete}(x.f) \rangle \Downarrow_i \langle \Sigma \backslash (l, f), h \backslash (l, f), \rho, \mathit{Cont} \rangle}$$

Open RHS - Assignment
$$\frac{v = [\![e]\!]_\rho \qquad \neg\mathsf{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x := e \rangle \Downarrow_i \langle \Sigma, h, \rho, \mathit{Err} \rangle}$$

Sequencing-1
$$\frac{\langle \Sigma, h, \rho, s_1 \rangle \Downarrow_i \langle \Sigma_1, h_1, \rho_1, \mathit{Err} \rangle}{\langle \Sigma, h, \rho, s_1; s_2 \rangle \Downarrow_i \langle \Sigma_2, h_2, \rho_2, \mathit{Err} \rangle}$$

Sequencing-2
$$\frac{\langle \Sigma, h, \rho, s_1 \rangle \Downarrow_i \langle \Sigma_1, h_1, \rho_1, \mathit{Ret}(v) \rangle}{\langle \Sigma, h, \rho, s_1; s_2 \rangle \Downarrow_i \langle \Sigma_2, h_2, \rho_2, \mathit{Ret}(v) \rangle}$$

Sequencing-3
$$\frac{\langle \Sigma, h, \rho, s_1 \rangle \Downarrow_i \langle \Sigma_1, h_1, \rho_1, \mathit{Cont} \rangle \qquad \langle \Sigma_1, h_1, \rho_1, s_2 \rangle \Downarrow_i \langle \Sigma_2, h_2, \rho_2, o \rangle}{\langle \Sigma, h, \rho, s_1; s_2 \rangle \Downarrow_i \langle \Sigma_2, h_2, \rho_2, o \rangle}$$

Return
$$\frac{[\![e]\!]_\rho = v}{\langle \Sigma, h, \rho, \mathsf{return}(e) \rangle \Downarrow_i \langle \Sigma, h, \rho, \mathit{Ret}(v) \rangle}$$

Open Function Call
$$\frac{[\![e_i]\!]_\rho = v_i|_{i=1}^n \qquad \exists_{j \in \{1,\dots,n\}} : \neg\mathsf{Closed}_\Sigma(v_j)}{\langle \Sigma, h, \rho, x := f(e_1, \dots, e_n) \rangle \Downarrow_i \langle \Sigma, h, \rho, \mathit{Err} \rangle}$$

Function Call
$$\frac{\begin{array}{c}[\![e_i]\!]_\rho = v_i|_{i=1}^n \qquad \mathsf{body}(f) = s \qquad \mathsf{params}(f) = x_i|_{i=1}^n \\ \langle \Sigma, h, [x_i \mapsto v_i|_{i=1}^n], s \rangle \Downarrow_i \langle \Sigma', h', \rho', \mathit{Ret}(v) \rangle \qquad \mathsf{Closed}_\Sigma(v_i)_{i=1}^n\end{array}}{\langle \Sigma, h, \rho, x := f(e_1, \dots, e_n) \rangle \Downarrow_i \langle \Sigma', h', \rho[x \mapsto v], \mathit{Cont} \rangle}$$

Figure 3.6: Big-Step semantics for statements - function call: $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho', o \rangle$

and of the heap typing environment and continues with the computation normally, which we represent using the continuation outcome Cont.

[Open RHS - Assignment] This rule is applied when the location of an open object is assigned to a variable. The rule evaluates the expression being assigned and checks that its value is open. Then, it generates an error using the error outcome Err.

[Sequencing-1] This rule is applied when the first step of sequential statement evaluates to an error. The rule only checks if the first step of the sequential statement leads to an error and immediately outputs the error (without executing the second statement).

[Sequencing-2] This rule is applied when the first step of sequential statement evaluates to a return. The rule only checks if the first step of the sequential statement leads to a return and immediately outputs the same return outcome (without executing the second statement).

[Sequencing-3] This rule is applied when the first step of the sequential statement evaluates to a continuation outcome, in which case the second statement must also be evaluated. The rule first evaluates the first element of the sequence, checking that its evaluation results in a continuation outcome. Then, the rule evaluates the second statement on the heap typing environment, heap, and store obtained from the evaluation of the first statement.

[Return] This rule is applied when evaluating a return statement. The rule simply evaluates the expression being returned, obtaining a value v and then generates the return outcome parameterized with v, $\mathit{Ret}(v)$.

[Open Function Call] This rule is applied when a function is called with an open argument. Calling a function with an open argument necessarily creates a second reference pointing to it (that being the formal parameter of the function to be called). Hence, calling a function with one or various open arguments violates the NAOO invariant. For this reason, such calls immediately generate an error outcome.

[FUNCTION CALL] This rule is applied when a function is called with closed arguments. The rule starts by evaluating the supplied arguments, obtaining a list of closed values $v_i|_{i=1}^{n}$. Then, the rule obtains the list of formal parameters of the function to be executed $x_i|_{i=1}^{n}$ and the statement s corresponding to the body of the function, and creates a new store mapping the formal parameters of the function to the supplied arguments, $[x_i \mapsto v_i|_{i=1}^{n}]$. Next, the rule executes the body of the function in the newly created store together with the original heap and heap typing environment, obtaining a final heap, heap typing environment, and returned value v. Finally, the returned value v is assigned to the program variable x.

Soundness of the Complete Type System We now prove that our full type system satisfies the Type Safety and Fault Avoidance properties with respect to the extended operational semantics.

Theorem 3 (Soundness - Type Safety)**.** *Let g be a function and Δ a typing context. Let h be a heap, ρ a store and s a statement. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$, $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho', o \rangle$, $h, \rho \vDash \Sigma, \Gamma$ and $\Delta \vdash p$. Then $h', \rho' \vDash \Sigma', \Gamma'$ and $o \vDash_{\Sigma'} \Delta_r(g)$.*

Proof. The proof follows by induction on the derivation of $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho', o \rangle$. Hence, assuming that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$ (hyp.1), $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho', o \rangle$ (hyp.2) and $h, \rho \vDash \Sigma, \Gamma$ (hyp.3), we have the following base cases:

[RETURN] Suppose that $s = \text{return}(e)$(hyp.4). We have that:

- $\frac{\Delta(g)=(\tau_1',\ldots,\tau_n')\rightarrow\tau_e \;\; \Gamma\vdash e:\tau_e}{g,\Delta\vdash\{\Gamma\}return\ e\{\Gamma\}}$ is applied. (1) - hyp.1 + hyp.4
- $\langle \Sigma, h, \rho, \text{return}(e) \rangle \Downarrow_i \langle \Sigma, h, \rho, Ret(v) \rangle$ (2) - hyp.2 + hyp.4
- $\Gamma' = \Gamma$ (3) - (1)
- $\Sigma' = \Sigma, h' = h, \rho' = \rho$ (4) - (2)
- $h', \rho' \vDash \Sigma', \Gamma'$ (5) - (3) + (4)
- $\Delta_r(g) = \tau_e$ (6) - (1)
- $\Gamma \vdash e : \tau_e$ (7) - (1)
- $[\![e]\!]_\rho = v$ (8) - (2)
- $v \vDash_{\Sigma'} \tau_e$ (9) - (7) + (8) + hyp.2 + Lemma WTE-Safety
- $v \vDash_{\Sigma'} \Delta_r(g)$ (10) - (6)

[FUNCTION CALL] Suppose that $s = x := f(e_1, ..., e_n)$ (hyp.4). We have that:

- $\frac{\Gamma\vdash e_i:\tau_i|_{i=1}^{n} \;\; \Delta(f)=(\tau_1,\ldots,\tau_n)\rightarrow\tau \;\; Closed(\tau_i)|_{i=1}^{n}}{g,\Delta\vdash\{\Gamma\}x:=f(e_i|_{i=1}^{n})\{\Gamma[x\mapsto\tau]\}}$ is applied. (1) - hyp.1 + hyp.4
- $\langle \Sigma, h, \rho, x := f(e_1, ..., e_n) \rangle \Downarrow_i \langle \Sigma'', h'', \rho[x \mapsto v], Cont \rangle$ (2) - hyp.2 + hyp.4
- $[\![e_i]\!]_\rho = v_i|_{i=1}^{n}$ (3) - (2)
- $\text{body}(f) = s$ (4) - (2)

- params$(f) = x_i|_{i=1}^{n}$ (5) - (2)
- $\langle \Sigma, h, [x_i \mapsto v_i|_{i=1}^{n}], s\rangle \Downarrow_i \langle \Sigma', h', \rho', Ret(v)\rangle$ (6) - (2)
- Closed$_\Sigma(v_i)_{i=1}^{n}$ (7) - (2)
- $\Gamma' = \Gamma[x \mapsto \tau]$ (8) - (1)
- $\Sigma' = \Sigma'', h' = h'', \rho' = \rho[x \mapsto v]$ (9) - (2)
- $h', \rho' \vDash \Sigma', \Gamma'$ (10) - hyp.3 + hyp.ind

The remaining cases are analogous to the ones presented in Subsection 3.3.4. □

Theorem 4 (Soundness - Fault Avoidance)**.** *Let g be a function and Δ a Typing Context. Let h be a heap, ρ a store and s a statement. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$ and $h, \rho \vDash \Sigma, \Gamma$ then it is not the case that $\langle \Sigma, h, \rho, s\rangle \Downarrow_i \langle \Sigma', h', \rho', Err\rangle$.*

Proof. Assuming that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$ (hyp.1) and $h, \rho \vDash \Sigma, \Gamma$ (hyp.2). Suppose also by contradiction that $\langle \Sigma, h, \rho, s\rangle \Downarrow_i \langle \Sigma', h', \rho', Err\rangle$ (hyp.3).

[Unclosed Function Argument] Suppose that $s = x := f(e_1, ..., e_n)$ (hyp.4). We have that:

- $\frac{\Gamma \vdash e_i : \tau_i|_{i=1}^{n}\ \ \Delta(f) = (\tau_1, \ldots, \tau_n) \to \tau\ \ Closed(\tau_i)|_{i=1}^{n}}{g, \Delta \vdash \{\Gamma\} x := f(e_i|_{i=1}^{n}) \{\Gamma[x \mapsto \tau]\}}$ is applied. (1) - hyp.1 + hyp.4
- $\Gamma \vdash e_i : \tau_i|_{i=1}^{n}$ (2) - (1)
- $Closed(\tau_i)|_{i=1}^{n}$ (3) - (1)
- $\exists_{v_i} :\ [\![e_i]\!]_\rho = v_i$ (4) - hyp.1 + hyp.2
- $\exists_{v_j} \neg$Closed$_\Sigma(v_j)$ (5) - (4) + hyp.3 + hyp.4
- $v_j \vDash_\Sigma \tau_j$ (6) - (2) + (4) + hyp.2 + Lemma WTE-Safety
- Closed$_\Sigma(v_j)$ (7) - (3) + (6) + Lemma closed values
- Contradiction (8) - (5) + (7)

□

Chapter 4

Small-Step Soundness

In this chapter we prove the soundness of our type system with respect to a small-step semantics of ECMA-SL. In Section 4.1, we introduce our small-step semantics for ECMA-SL. Afterwards, in Sections 4.2 and 4.3 we prove preservation and progress by resorting to the lemmas introduced in these sections and in the previous chapter.

In order to simplify the exposition, as in the previous chapter, the semantics of ECMA-SL that we first present does not model function calls. Thus, in Section 3.5 we extend the semantics to cater for this aspect.

4.1 Small-Step Semantics

In this section we define a small-step semantics for ECMA-SL statements, ignoring for now function calls. Once again, we rely on the semantics for expressions presented in Figure 3.2, using the notation $[\![e]\!]_\rho = v$ to mean that the evaluation of the expression e in the store ρ results in the value v.

The small-step semantics judgements for statements are of the form $\langle \Sigma, h, \rho, s\rangle \rightarrow_i \langle \Sigma', h', \rho', s'\rangle$, meaning that the evaluation of the statement s in the heap h and store ρ leads to a statement s' in the heap h' and store ρ'. Similarly to our big-step semantics, and in order to reason about the types of the objects in the heap, we have to instrument the semantics to keep track of the types of the objects created at runtime. To this end, the semantic judgement for statements additionally include the initial and final heap typing environments, respectively Σ and Σ'.

Our small-step semantics is equivalent to the big-step semantics provided in the previous chapter. Put formally, for every heap typing environments Σ and Σ', heaps h and h', stores ρ and ρ', and statement s, it holds that:

$$\langle \Sigma, h, \rho, s\rangle \Downarrow_i \langle \Sigma', h', \rho'\rangle \iff \langle \Sigma, h, \rho, s\rangle \rightarrow_i^* \langle \Sigma', h', \rho', \mathsf{skip}\rangle \tag{4.1}$$

where: we use $\rightarrow_i^*$ to denote the reflexive-transitive closure of $\rightarrow_i$. From the equivalence result, it follows that the small-step semantics also enforces the *no aliasing for open objects* (NAOO) invariant. Hence, we do not present a formal proof of this fact as it would be essentially a replay of the proof provided in Subsection 3.3.1.

Sequencing Skip
$$\langle \Sigma, h, \rho, \mathsf{skip}; s\rangle \rightarrow_i \langle \Sigma, h, \rho, s\rangle$$

Assignment
$$\frac{v = [\![e]\!]_\rho \qquad \mathsf{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x := e\rangle \rightarrow_i \langle \Sigma, h, \rho[x \mapsto v], \mathsf{skip}\rangle}$$

Assignment From field
$$\frac{[\![e]\!]_\rho = l \qquad h(l, f) = v \qquad \mathsf{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x := e.f\rangle \rightarrow_i \langle \Sigma, h, \rho[x \mapsto v], \mathsf{skip}\rangle}$$

New Object
$$\frac{l \notin dom(h) \qquad \Sigma' = \Sigma[l \mapsto \{\}^\circ] \qquad h' = h[l \mapsto \{\}]}{\langle \Sigma, h, \rho, x := \{\}\rangle \rightarrow_i \langle \Sigma', h', \rho[x \mapsto l], \mathsf{skip}\rangle}$$

field Assignment Open
$$\frac{[\![e]\!]_\rho = v \qquad [\![x]\!]_\rho = l \qquad \tau = Type_\Sigma(v) \qquad \lfloor \Sigma(l) \rfloor = \circ \qquad \Sigma' = \Sigma[l \mapsto \Sigma(l)[f \mapsto \tau]] \qquad \mathsf{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x.f := e\rangle \rightarrow_i \langle \Sigma', h[(l, f) \mapsto v], \rho, \mathsf{skip}\rangle}$$

field Assignment Close
$$\frac{[\![e]\!]_\rho = v \qquad [\![x]\!]_\rho = l \qquad \Sigma(l, f) = Type_\Sigma(v) \qquad \lfloor \Sigma(l) \rfloor = \bullet \qquad (l, f) \in dom(h) \qquad \mathsf{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x.f := e\rangle \rightarrow_i \langle \Sigma, h[(l, f) \mapsto v], \rho, \mathsf{skip}\rangle}$$

field Delete
$$\frac{[\![x]\!]_\rho = l \qquad \lfloor \Sigma(l) \rfloor = \circ}{\langle \Sigma, h, \rho, \mathsf{delete}(x.f)\rangle \rightarrow_i \langle \Sigma \backslash (l, f), h \backslash (l, f), \rho, \mathsf{skip}\rangle}$$

Commit
$$\frac{[\![x]\!]_\rho = l \qquad \lfloor \Sigma(l) \rfloor = \circ}{\langle \Sigma, h, \rho, commit(x)\rangle \rightarrow_i \langle \Sigma[l \mapsto \Sigma(l)^\bullet], h, \rho, \mathsf{skip}\rangle}$$

If-True
$$\frac{[\![e]\!]_\rho = true}{\langle \Sigma, h, \rho, \mathsf{if}(e)\{s_1\} \text{ else } \{s_2\}\rangle \rightarrow_i \langle \Sigma, h, \rho, s_1\rangle}$$

If-False
$$\frac{[\![e]\!]_\rho = false}{\langle \Sigma, h, \rho, \mathsf{if}(e)\{s_1\} \text{ else } \{s_2\}\rangle \rightarrow_i \langle \Sigma, h, \rho, s_2\rangle}$$

While
$$\langle \Sigma, h, \rho, \mathsf{while}(e)\{s\}\rangle \rightarrow_i \langle \Sigma, h, \rho, \mathsf{if}(e)\{s; \mathsf{while}(e)\{s\}\} \text{ else } \{\mathsf{skip}\}\rangle$$

Sequencing Composition
$$\frac{\langle \Sigma, h, \rho, s_1\rangle \rightarrow_i \langle \Sigma_2, h_2, \rho_2, s_1'\rangle}{\langle \Sigma, h, \rho, s_1; s_2\rangle \rightarrow_i \langle \Sigma_2, h_2, \rho_2, s_1'; s_2\rangle}$$

Figure 4.1: Small-Step semantics for statements: $\langle \Sigma, h, \rho, s\rangle \rightarrow_i \langle \Sigma', h', \rho', s'\rangle$

The semantic rules are given in Figure 4.1 and explained below. Unsurprisingly, the small-step rules are analogous to the big-step rules given in the previous chapter. In fact, the set of rules that lead to a skip statement almost exactly coincide with the corresponding big-step rules. The major differences appear in the rules associated with compound statements: if, while, and sequence. For these statements, the corresponding small-step rules perform a single computation step, generating the statement to be executed next, while the big-step rules capture their complete evaluation.

[Sequencing Skip] This rule is applied when there is a sequence statement which has as first element a skipstatement. The rule simply proceeds to the second statement of the sequencing.

[Var Assignment] This rule is used when a closed value is being assigned to a program variable. The rule first evaluates the expression e obtaining the value v, which is then assigned to the variable x in the store ρ. The rule proceeds to a skip statement.

[Field Lookup] This rule is applied when assigning a closed value from an object field to a program variable. The rule first evaluates the expression e obtaining the object location l. Then, it obtains the value v associated with the field f in the object pointed to by l and updates the value of the variable x to v in the store ρ. The rule proceeds to a skip statement.

[New Object] This rule is applied when assigning a new object to a variable x. The rule finds a location l which is not in the domain of the heap and adds such location to both heap and heap typing environment, mapping it to an empty object and to an empty open object type, respectively. The rule also maps x to l in the store and proceeds to a skip statement.

[FIELD ASSIGNMENT - OPEN] This rule is applied when assigning a closed value to a field of an open object. The rule first evaluates the expression e obtaining the value v, and the location l of the program variable x. Then, it updates the field f being modified and its type in the heap and heap typing environment, respectively. The rule proceeds to a skip statement.

[FIELD ASSIGNMENT - CLOSE]This rule is applied when assigning a closed value to a field of a close object. The rule first evaluates the expression e obtaining the value v, and the location l of the program variable x. Then, it verifies if the already existing field f of l has the same type as v and, if so, it updates the value of the field f to v in the heap h. The rule proceeds to a skip statement.

[FIELD DELETE] This rule is applied when deleting a field from an open object. The rule first obtains the location l of the program variable x. Then it removes the pair (l, f) from the domain of both heap and heap typing environment. The rule proceeds to a skip statement.

[COMMIT] This rule is applied when closing an open object. The rule first obtains the location l of the program variable x. Afterwards it updates the type associated with the location l to a closed type in the store typing environment Σ. The rule proceeds to a skip statement.

[IF-TRUE] This rule is applied when evaluating an if statement and its associated boolean expression evaluates to $true$. The rule proceeds to the then branch statement.

[IF-FALSE] This rule is applied when evaluating an if statement and its associated boolean expression evaluates to $false$. The rule proceeds to the then branch statement.

[WHILE] This rule is applied when evaluating a $while$ statement. The rule proceeds to the conditional statement with the same boolean expression as the while, as then branch statement the body of the while, and as else branch statement skip.

[SEQUENCING COMPOSITION] This rule is used to evaluate a sequence of two statements $s_1; s_2$. The rule delegates its job to a support transition which goes from the statements s_1 to s_1'. The main rule then proceeds with the resulting heap typing environment Σ_2, heap h_2, store ρ_2 and statement $s_1'; s_2$.

4.2 Soundness - Preservation

In this section we prove that the proposed type system satisfies the *preservation property* with respect to the small-step operational semantics defined in Section 4.1. Essentially, *preservation* states that the small-step operational semantics *preserves* satisfiability, meaning that if one starts executing a statement in a state that satisfies a given heap and typing environment, all the intermediate states will satisfy their corresponding heaps and typing environment. Hence, if a statement s is typable with respect to a given function g, typing context Δ and store typing environments Γ and Γ_f, i.e. $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma_f\}$, and if one performs one execution step of s in a state (h, ρ) such that $h, \rho \vDash \Sigma, \Gamma$, for a given heap typing environment Σ, obtaining the statement to be executed next, s', the state (h', ρ'), and the heap typing environment Σ'. Then, there exists a variable typing environment Γ', such that $h', \rho' \vDash \Sigma', \Gamma'$ and $g, \Delta \vdash \{\Gamma'\}\ s'\ \{\Gamma_f\}$. Put formally:

$$\begin{gathered} g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma_f\}\ \wedge\ h, \rho \vDash \Sigma, \Gamma\ \wedge\ \langle \Sigma, h, \rho, s\rangle \rightarrow_i \langle \Sigma', h', \rho', s'\rangle \\ \Longrightarrow \\ \exists \Gamma' : h', \rho' \vDash \Sigma', \Gamma'\ \wedge\ g, \Delta \vdash \{\Gamma'\}\ s'\ \{\Gamma_f\} \end{gathered} \tag{4.2}$$

Below, we introduce Theorem 5 stating that the proposed type system satisfies the Preservation property. Similarly to the type safety theorem, the preservation theorem uses the lemmas introduced in Section 3.3.

Theorem 5 (Preservation)**.** *Let g be a function and Δ a Typing Context. Let Σ and Σ' heap typing environments, Γ, Γ' and Γ_f store typing environments, h and h' heaps, ρ and ρ' stores and s and s' statements. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma_f\}$, $\langle \Sigma, h, \rho, s\rangle \rightarrow_i \langle \Sigma', h', \rho', s'\rangle$ and $h, \rho \vDash \Sigma, \Gamma$. Then $\exists \Gamma' : h', \rho' \vDash \Sigma', \Gamma'\ \wedge\ g, \Delta \vdash \{\Gamma'\}\ s'\ \{\Gamma_f\}$*

Proof. The proof follows by induction on the derivation of $\langle \Sigma, h, \rho, s\rangle \rightarrow_i \langle \Sigma', h', \rho', s'\rangle$. Hence, assuming that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma_f\}$ (hyp.1), $\langle \Sigma, h, \rho, s\rangle \rightarrow_i \langle \Sigma', h', \rho', s'\rangle$ (hyp.2) and $h, \rho \vDash \Sigma, \Gamma$ (hyp.3), we have the following base cases:

[ASSIGNMENT] Suppose that $s = x := e$(hyp.4). We have that:

- $\frac{\Gamma \vdash e : \tau_e\ Closed(\tau_e)}{g, \Delta \vdash \{\Gamma\}\ x := e\ \{\Gamma[x \mapsto \tau_e]\}}$ is applied (1) - hyp.1 + hyp.4
- $\Sigma' = \Sigma$, $h' = h$, $\rho' = \rho[x \mapsto [\![e]\!]_\rho]$, $s' =$ skip (2) - hyp.2 + hyp.4
- $\Gamma \vdash e : \tau_e$ (3) - (1)
- $[\![e]\!]_\rho \vDash_\Sigma \tau_e$ (4) - (2) + (3) + hyp.3 + Lemma WTE-Safety
- $\Gamma_f = \Gamma[x \mapsto \tau_e]$ (5) - (1)
- Taking $\Gamma' = \Gamma_f = \Gamma[x \mapsto \tau_e]$ (6) - (4)
- $h' \vDash \Sigma'$ (7) - (2) + hyp.3
- $\forall_{y \in dom(\rho), y \neq x}\ \Gamma(y) = \Gamma'(y)$ e $\rho(y) = \rho'(y)$ (8) - (2) + (6)
- Se $y = x$ então $\rho'(x) = [\![e]\!]_\rho$ e $\rho'(y) \vDash_\Sigma \Gamma'(y)$ (9) - (2) + (3) + (6) + Lemma WTE-Safety
- $\forall_{y \in dom(\Gamma')}\ \rho'(y) \vDash_\Sigma \Gamma'(y)$ (10) - (8) + (9) + hyp.3
- $\rho' \vDash_{\Sigma'} \Gamma'$ (11) - (2) + (10)
- $h', \rho' \vDash \Sigma', \Gamma'$ (12) - (7) + (11)
- $g, \Delta \vdash \{\Gamma'\}\ s'\ \{\Gamma_f\}$ (10) - (2) + (6)

[ASSIGNMENT FROM FIELD] Suppose that $x := e.f$(hyp.4). We have that:

- $\frac{\Gamma \vdash e : \{f_i : \tau_i|_{i=1}^{k}, f : \tau\}^*\ Closed(\tau)}{g, \Delta \vdash \{\Gamma\} x := e.f \{\Gamma[x \mapsto \tau]\}}$ is applied (1) - hyp.1 + hyp.4

- $\Sigma' = \Sigma$, $h' = h$, $\rho' = \rho[x \mapsto h([\![e]\!]_\rho, f)]$, $s' = \mathsf{skip}$ (2) - hyp.2 + hyp.4
- $\Gamma \vdash e : \Gamma \vdash e : \{f_i : \tau_i|_{i=1}^{k}, f : \tau\}^*$ (3) - (1)
- $h([\![e]\!]_\rho, f) \vDash_\Sigma \tau$ (4) - (2) + (3) + hyp.3 + Lemma WTE-Safety
- $\Gamma_f = \Gamma[x \mapsto \tau]$ (5) - (1)
- Taking $\Gamma' = \Gamma_f = \Gamma[x \mapsto \tau]$ (6) - (4)
- $h' \vDash \Sigma'$ (7) - (2) + hyp.3
- $\forall_{y \in dom(\rho), y \neq x}\ \Gamma(y) = \Gamma'(y)$ e $\rho(y) = \rho'(y)$ (8) - (2) + (6)
- Se $y = x$ então $\rho'(x) = h([\![e]\!]_\rho, f)$ e $\rho'(y) \vDash_\Sigma \Gamma'(y)$ (9) - (2) + (4) + (6)
- $\forall_{y \in dom(\Gamma')}\ \rho'(y) \vDash_\Sigma \Gamma'(y)$ (10) - (8) + (9) + hyp.3
- $\rho' \vDash_{\Sigma'} \Gamma'$ (11) - (2) + (10)
- $h', \rho' \vDash \Sigma', \Gamma'$ (12) - (7) + (11)
- $g, \Delta \vdash \{\Gamma'\}\ s'\ \{\Gamma_f\}$ (13) - (2) + (6)

[FIELD ASSIGNMENT OPEN] Suppose that $s = x.f := e$ (hyp.4) and $\lfloor \Sigma([\![x]\!]_\rho) \rfloor = \circ$ (hyp.5).

- $\langle \Sigma, h, \rho, x.f := e \rangle \to_i \langle \Sigma', h[(l, f) \mapsto v], \rho, \mathsf{skip} \rangle$ (1) - hyp.2 + hyp.4 + hyp.5
- $[\![x]\!]_\rho = l$ (2) - (1)
- $[\![e]\!]_\rho = v$ (3) - (1)
- $\tau = Type_\Sigma(v)$ (4) - (1)
- $\Sigma' = \Sigma[l \mapsto \Sigma(l)[f \mapsto \tau]], h' = h[(l, f) \mapsto v], \rho' = \rho, s' = \mathsf{skip}$ (5) - (1)
- $\mathsf{Closed}_\Sigma(v)$ (6) - (1)
- $l \vDash_\Sigma \Gamma(x)$ (7) - hyp.3 + (2) + Lemma WTE-Safety
- $\lfloor \Gamma(x) \rfloor = \circ$ (8) - hyp.5 + (7)

We have two cases to consider:

Case $(l, f) \notin dom(h)$ (hyp.6):

- $f \notin dom(\Sigma(l))$ (9.1.1) - hyp.3 + hyp.6
- $f \notin dom(\Gamma(x))$ (9.1.2) - (9.1.1)
- $\frac{\Gamma \vdash e : \tau_e \ \Gamma(x) = \{f_i : \tau_i|_{i=1}^{k}\}^\circ, \forall_{i \in \{1, \ldots, k\}} f \neq f_i}{g, \Delta \vdash \{\Gamma\} x.f := e \{\Gamma[x \mapsto \{f_i : \tau_i|_{i=1}^{k}, f : \tau_e\}^\circ]\}}$ is applied (9.1.3) - (8) + (9.1.2) + hyp.1
- $\Gamma \vdash e : \tau_e$ (9.1.4) - (9.1.3)
- $\Gamma(x) = \{f_i : \tau_i|_{i=1}^{k}\}^\circ$ (9.1.5) - (9.1.3)
- $\Gamma_f = \Gamma[x \mapsto \{f_i : \tau_i|_{i=1}^{k}, f : \tau_e\}^\circ]$ (9.1.6) - (9.1.3)
- $v \vDash_\Sigma \tau_e$ (9.1.7) - (3) + (9.1.4) + hyp.3 + Lemma WTE-Safety
- $\tau_e = \tau$ (9.1.9) - (9.1.7) + (4) + Lemma Satisfaction Uniqueness
- $h' \vDash \Sigma'$ (9.1.10) - (5) + (6) + (9.1.8) + hyp.3 + hyp.5 + $NAOO$ + Lemma Heap Update
- $\Gamma_f = \Gamma[x \mapsto \Gamma(x)[f \mapsto \tau]]$ (9.1.11) - (9.1.5) + (9.1.6) + (9.1.9)
- Taking $\Gamma' = \Gamma_f = \Gamma[x \mapsto \Gamma(x)[f \mapsto \tau]]$ (9.1.12) - (9.1.11)

- $\rho' \vDash_{\Sigma'} \Gamma'$ (9.1.13) - (5) + (6) + (9.1.12) + hyp.3 + hyp.5 + $NAOO_2$ + Lemma Store Update
- $h', \rho' \vDash \Sigma', \Gamma'$ (9.1.13) - (9.1.10) + (9.1.12)
- $g, \Delta \vdash \{\Gamma'\}\ s'\ \{\Gamma_f\}$ (9.1.14) - (5) + (9.1.11)

Case $(l, f) \in dom(h)$ (hyp.6):

- $f \in dom(\Sigma(l))$ (9.2.1) - hyp.3 + hyp.6
- $f \in dom(\Gamma(l))$ (9.2.2) - (9.2.1)
- $\frac{\Gamma \vdash e:\tau_e \ \Gamma(x)=\{f_i:\tau_i|_{i=1}^{k}\}^\circ, \exists_{j\in\{1,\ldots,k\}} f=f_j}{g,\Delta\vdash\{\Gamma\}x.f:=e\{\Gamma[x\mapsto\{f_i:\tau_i|_{i=1,i\neq j}^{k}, f:\tau_e\}^\circ]\}}$ is applied. (9.1.3) - (8) + (9.1.2) + hyp.1
- $\Gamma \vdash e : \tau_e$ (9.1.4) - (9.1.3)
- $\Gamma(x) = \{f_i : \tau_i|_{i=1}^{k}\}^\circ$ (9.1.5) - (9.1.3)
- $\Gamma_f = \Gamma[x \mapsto \{f_i : \tau_i|_{i=1,i\neq j}^{k}, f : \tau_e\}^\circ]$ (9.1.6) - (9.1.3)
- $v \vDash_\Sigma \tau_e$ (9.1.7) - (3) + (9.1.4) + hyp.3 + Lemma WTE-Safety
- $v \vDash_\Sigma \tau$ (9.1.8) - (4)
- $\tau_e = \tau$ (9.1.9) - (9.1.7) + (9.1.8) + Lemma Satisfaction Uniqueness
- $h' \vDash \Sigma'$ (9.1.10) - (5) + (6) + (9.1.8) + hyp.3 + hyp.5 + $NAOO$ + Lemma Heap Update
- $\Gamma'_f = \Gamma[x \mapsto \Gamma(x)[f \mapsto \tau]]$ (9.1.11) - (9.1.5) + (9.1.6) + (9.1.9)
- Taking $\Gamma' = \Gamma_f = \Gamma[x \mapsto \Gamma(x)[f \mapsto \tau]]$ (9.1.12) - (9.1.11)
- $\rho' \vDash_{\Sigma'} \Gamma'$ (9.1.13) - (5) + (6) + (9.1.12) + hyp.3 + hyp.5 + $NAOO_2$ + Lemma Store Update
- $h', \rho' \vDash \Sigma', \Gamma'$ (9.1.14) - (9.1.10) + (9.1.13)
- $g, \Delta \vdash \{\Gamma'\}\ s'\ \{\Gamma_f\}$ (9.1.14) - (5) + (9.1.11)

[FIELD ASSIGNMENT CLOSE] Suppose that $s = x.f := e$ (hyp.4) and $\lfloor\Sigma([\![x]\!]_\rho)\rfloor = \bullet$ (hyp.5). We have that:

- $\frac{\Gamma\vdash e:\tau_e \ \Gamma(x)=\{\ldots,f:\tau_f,\ldots\}^\bullet \ \tau_e=\tau_f}{g,\Delta\vdash\{\Gamma\}x.f:=e\{\Gamma\}}$ is applied (1) - hyp.1 + hyp.4 + hyp.5
- $\Gamma \vdash e : \tau_e$ (2) - (1)
- $\Gamma(x) = \{..., f : \tau_f, ...\}^\bullet$ (3) - (1)
- $\tau_e = \tau_f$ (4) - (1)
- $\langle\Sigma, h, \rho, x.f := e\rangle \to_i \langle\Sigma, h[(l, f) \mapsto v], \rho, \mathsf{skip}\rangle$ (5) - hyp.2 + hyp.4 + hyp.5
- $[\![e]\!]_\rho = v$ (6) - (5)
- $[\![x]\!]_\rho = l$ (7) - (5)
- $\Sigma(l, f) = Type_\Sigma(v)$ (8) - (5)
- $h(l, f) = v_f$ (9) - (5)
- $\Gamma_f = \Gamma$ (10) - (1)
- Making $\Gamma' = \Gamma_f = \Gamma$ (11) - (10)
- $\Sigma' = \Sigma, h' = h[(l, f) \mapsto v], \rho' = \rho, s' = \mathsf{skip}$ (12) - (5)
- $\rho' \vDash_{\Sigma'} \Gamma'$ (13) - (11) + (12) + hyp.3

We now have two situations to consider to show that $\forall_{l\in dom(\Sigma)} h'(l) \vDash \Sigma'(l)$:

- If $\hat{l} \neq l$ then $h'(\hat{l}) = h(\hat{l})$ therefore $h'(\hat{l}) \vDash \Sigma'(\hat{l})$ (14.1) - (12)
- If $\hat{l} = l$ we have that: (14.2)
  - $\forall_{g \in dom(h) \setminus f} : h'(\hat{l}, g) = h(\hat{l}, g)$ (14.2.2) - (12)
  - $\forall_{g \in dom(h) \setminus f} : h'(\hat{l}, g) \vDash_{\Sigma'} \Sigma'(\hat{l}, g)$ (14.2.3) - (12) + (14.2.2)
  - $v \vDash_{\Sigma} \tau_e$ (14.2.4) - (2) + (6)
  - $v \vDash_{\Sigma'} \tau_e$ (14.2.5) - (8) + (14.2.4)
  - $v \vDash_{\Sigma'} \tau_f$ (14.2.6) - (4) + (14.2.5)
  - $h'(\hat{l}, f) \vDash_{\Sigma'} \Sigma'(\hat{l}, f)$ (14.2.7) - (3) + (12) + (14.2.6)
  - $h'(\hat{l}) \vDash \Sigma'(\hat{l})$ (14.2.8) - (14.2.1) + (14.2.3) + (14.2.7)

Therefore:

- $h' \vDash \Sigma'$ (15) - (14.1) + (14.2)
- $h', \rho' \vDash \Sigma', \Gamma'$ (16) - (13) + (15)
- $g, \Delta \vdash \{\Gamma'\}\ s'\ \{\Gamma_f\}$ (17) - (11) + (12)

[COMMIT] Suppose that $s = commit(x)$(hyp.4). We have that:

- $\frac{\lfloor \Gamma(x) \rfloor = \circ}{g, \Delta \vdash \{\Gamma\}\ commit(x)\ \{\Gamma[x \mapsto \Gamma(x)^{\bullet}]\}}$ is applied. (1) - hyp.1 + hyp.4
- $\langle \Sigma, h, \rho, commit(x) \rangle \Downarrow_i \langle \Sigma[l \mapsto \Sigma(l)^{\bullet}], h, \rho \rangle$ (2) - hyp.2 + hyp.4
- $[\![x]\!]_{\rho} = l$ (3) - (2)
- $\lfloor \Sigma(l) \rfloor = \circ$ (4) - (2)
- $\Gamma_f = \Gamma[x \mapsto \Gamma(l)^{\bullet}]$ (5) - (1)
- Making $\Gamma' = \Gamma_f = \Gamma[x \mapsto \Gamma(l)^{\bullet}]$ (6) - (5)
- $\Sigma' = \Sigma[l \mapsto \Sigma(l)^{\bullet}], h' = h, \rho' = \rho, s' = \text{skip}$ (7) - (2)
- $h' \vDash \Sigma'$ (8) - (4) + (7) + hyp.3 + Lemma Heap Close
- $\rho' \vDash_{\Sigma'} \Gamma'$ (8) - (3) + (4) + (6) + (7) + hyp.3 + Lemma Store Close
- $h', \rho' \vDash \Sigma', \Gamma'$ (9) - (7) + (8)
- $g, \Delta \vdash \{\Gamma'\}\ s'\ \{\Gamma_f\}$ (10) - (5) + (6)

[NEW OBJECT] Suppose that $s = x := \{\}$(hyp.4). We have that:

- $g, \Delta \vdash \{\Gamma\} x := \{\} \{\Gamma[x \mapsto \{\}^{\circ}]\}$ is applied. (1) - hyp.1 + hyp.4
- $\langle \Sigma, h, \rho, x := \{\} \rangle \rightarrow_i \langle \Sigma', h', \rho[x \mapsto l], \text{skip} \rangle$ (2) - hyp.2 + hyp.4
- $l \notin dom(h)$ (3) - (2)
- $\Gamma_f = \Gamma[x \mapsto \{\}^{\circ}]$ (4) - (1)
- Making $\Gamma' = \Gamma_f = \Gamma[x \mapsto \{\}^{\circ}]$ (5) - (4)
- $\Sigma' = \Sigma[l \mapsto \{\}^{\circ}], h' = h[l \mapsto \{\}], \rho' = \rho[x \mapsto l], s' = \text{skip}$ (6) - (2)
- $h' \vDash \Sigma'$ (7) - (3) + (6) + hyp.3 + Lemma Heap New

- $\rho' \models_{\Sigma'} \Gamma'$ (8) - (3) + (5) + (6) + hyp.3 + Lemma Store New
- $h', \rho' \models \Sigma', \Gamma'$ (9) - (7) + (8)
- $g, \Delta \vdash \{\Gamma'\}\ s'\ \{\Gamma_f\}$ (10) - (5) + (6)

[FIELD DELETE] Suppose that $s = \text{delete}(x)$(hyp.4). We have that:

- $\frac{\Gamma(x)=\{f_i:\tau_i|_{i=1}^{k}\}^{\circ},\exists_{j\in\{1,\ldots,k\}}f=f_j}{g,\Delta\vdash\{\Gamma\}delete\ x.f\{\Gamma[x\mapsto\{f_i:\tau_i|_{i=1,i\neq j}^{k}\}^{\circ}]\}}$ is applied. (1) - hyp.1 + hyp.4
- $\langle \Sigma, h, \rho, \text{delete}(x.f)\rangle \Downarrow_i \langle \Sigma\backslash(l, f), h\backslash(l, f), \rho\rangle$ (2) - hyp.2 + hyp.4
- $[\![x]\!]_\rho = l$ (3) - (2)
- $\Gamma_f = \Gamma\backslash(x, f)$ (4) - (2)
- Making $\Gamma' = \Gamma_f = \Gamma\backslash(x, f)$ (5) - (2)
- $\Sigma' = \Sigma\backslash(l, f), h' = h\backslash(l, f), \rho' = \rho, s' = \text{skip}$ (6) - (2)
- $h' \models \Sigma'$ (7) - (6) + hyp.1 + Lemma Heap Delete
- $\rho' \models_{\Sigma'} \Gamma'$ (8) - (5) + (6) + hyp.1 + Lemma Store Delete
- $h', \rho' \models \Sigma', \Gamma'$ (9) - (7) + (8)
- $g, \Delta \vdash \{\Gamma'\}\ s'\ \{\Gamma_f\}$ (10) - (5) + (6)

[IF-TRUE] It follows that $s = \text{if}(e)\{s_1\}$ else $\{s_2\}$ (hyp.4) and $[\![e]\!]_\rho = true$ (hyp.5). We have that:

- $\frac{g,\Delta\vdash e:bool\ \ g,\Delta\vdash\{\Gamma\}s_1\{\Gamma_1\}\ \ g,\Delta\vdash\{\Gamma\}s_2\{\Gamma_2\}}{g,\Delta\vdash\{\Gamma\}if(e)\{s_1\}else\{s_2\}\{\Gamma_1\sqcap\Gamma_2\}}$ is applied. (1) - hyp.1 + hyp.4
- $g, \Delta \vdash \{\Gamma\}s_1\{\Gamma_1\}$ (2) - (1)
- $\Gamma_f = \Gamma_1 \sqcap \Gamma_2$ (3) - (1)
- $\langle \Sigma, h, \rho, \text{if}(e)\{s_1\}$ else $\{s_2\}\rangle \rightarrow_i \langle \Sigma, h, \rho, s_1\rangle$ (4) - hyp.2 + hyp.4 + hyp.5
- $\Sigma' = \Sigma, h' = h, \rho' = \rho, s' = s_1$ (5) - (2)
- Making $\Gamma' = \Gamma$ (6)
- $h', \rho' \models \Sigma', \Gamma'$ (7) - (5) + (6) + hyp.3
- $g, \Delta \vdash \{\Gamma\}s_1\{\Gamma_1 \sqcap \Gamma_2\}$ (8) - (2) + Lemma Weakening
- $g, \Delta \vdash \{\Gamma'\}s_1\{\Gamma_f\}$ (9) - (6) + (8)

[IF-FALSE] It follows that $s = \text{if}(e)\{s_1\}$ else $\{s_2\}$ (hyp.4) and $[\![e]\!]_\rho = false$ (hyp.5). We have that:

- $\frac{g,\Delta\vdash e:bool\ \ g,\Delta\vdash\{\Gamma\}s_1\{\Gamma_1\}\ \ g,\Delta\vdash\{\Gamma\}s_2\{\Gamma_2\}}{g,\Delta\vdash\{\Gamma\}if(e)\{s_1\}else\{s_2\}\{\Gamma_1\sqcap\Gamma_2\}}$ is applied. (1) - hyp.1 + hyp.4
- $g, \Delta \vdash \{\Gamma\}s_2\{\Gamma_2\}$ (2) - (1)
- $\Gamma_f = \Gamma_1 \sqcap \Gamma_2$ (3) - (1)
- $\langle \Sigma, h, \rho, \text{if}(e)\{s_1\}$ else $\{s_2\}\rangle \rightarrow_i \langle \Sigma, h, \rho, s_2\rangle$ (4) - hyp.2 + hyp.4 + hyp.5
- $\Sigma' = \Sigma, h' = h, \rho' = \rho, s' = s_2$ (5) - (2)
- Making $\Gamma' = \Gamma$ (6)
- $h', \rho' \models \Sigma', \Gamma'$ (7) - (5) + (6) + hyp.3
- $g, \Delta \vdash \{\Gamma\}s_2\{\Gamma_1 \sqcap \Gamma_2\}$ (8) - (2) + Lemma Weakening
- $g, \Delta \vdash \{\Gamma'\}s_2\{\Gamma_f\}$ (9) - (6) + (8)

[WHILE] Suppose that $s = \mathsf{while}(e)\{s\}$ (hyp.4). We have that:

- $\frac{g,\Delta\vdash e:bool \quad g,\Delta\vdash\{\Gamma\}s\{\Gamma\}}{g,\Delta\vdash\{\Gamma\}while(e)\{s\}\{\Gamma\}}$ is applied. (1) - hyp.1 + hyp.4
- $g, \Delta \vdash \{\Gamma\}s\{\Gamma\}$ (2) - (1)
- $g, \Delta \vdash e : bool$ (3) - (1)
- By derivation, $g, \Delta \vdash \{\Gamma\}\ \mathsf{if}(e)\{s; \mathsf{while}(e)\{s\}\}\ \mathsf{else}\ \{\mathsf{skip}\}\ \{\Gamma\}$ (4) - (2) + (3) + hyp.1

$$\frac{g, \Delta \vdash e : bool \qquad \dfrac{g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma\} \qquad g, \Delta \vdash \{\Gamma\}\ \mathsf{while}(e)\{s\}\ \{\Gamma\}}{g, \Delta \vdash \{\Gamma\}\ s; \mathsf{while}(s)\{s\}\ \{\Gamma\}} \qquad g, \Delta \vdash \{\Gamma\}\ \mathsf{skip}\ \{\Gamma\}}{g, \Delta \vdash \{\Gamma\}\ \mathsf{if}(e)\{s; \mathsf{while}(e)\{s\}\}\ \mathsf{else}\ \{\mathsf{skip}\}\ \{\Gamma\}}$$

- $\langle \Sigma, h, \rho, \mathsf{while}(e)\{s\}\rangle \to_i \langle \Sigma, h, \rho, \mathsf{if}(e)\{s; \mathsf{while}(e)\{s\}\}\ \mathsf{else}\ \{\mathsf{skip}\}\rangle$ (5) - hyp.2 + hyp.4
- $\Sigma' = \Sigma, h' = h, \rho' = \rho, s' = \mathsf{if}(e)\{s; \mathsf{while}(e)\{s\}\}\ \mathsf{else}\ \{\mathsf{skip}\}$ (6) - (5)
- $\Gamma_f = \Gamma$ (7) - (1)
- Making $\Gamma' = \Gamma$ (8)
- $h', \rho' \vDash \Sigma', \Gamma'$ (9) - (7) + (8) + hyp.3
- $g, \Delta \vdash \{\Gamma\}\ \mathsf{if}(e)\{s; \mathsf{while}(e)\{s\}\}\ \mathsf{else}\ \{\mathsf{skip}\}\ \{\Gamma\}$ (10) - (4) + (7) + (8)

[SEQUENCING] Suppose that $s = s_1; s_2$ (hyp.4). We have that:

- $\frac{g,\Delta\vdash\{\Gamma\}s_1\{\Gamma_1\} \quad g,\Delta\vdash\{\Gamma_1\}s_2\{\Gamma_2\}}{g,\Delta\vdash\{\Gamma\}s_1;s_2\{\Gamma_2\}}$ is applied. (1) - hyp.1 + hyp.4
- $g, \Delta \vdash \{\Gamma\}s_1\{\Gamma_1\}$ (2) - (1)
- $g, \Delta \vdash \{\Gamma_1\}s_2\{\Gamma_2\}$ (3) - (1)
- $\langle \Sigma, h, \rho, s_1; s_2\rangle \to_i \langle \Sigma_1, h_1, \rho_1, s_1'; s_2\rangle$ (4) - hyp.2 + hyp.4
- $\langle \Sigma, h, \rho, s_1\rangle \to_i \langle \Sigma_1, h_1, \rho_1, s_1'\rangle$ (5) - (4)
- $\Sigma' = \Sigma_1, h' = h_1, \rho' = \rho_1, s' = s_1'; s_2$ (6) - (4)
- $\exists \Gamma_1' : h_1, \rho_1 \vDash \Sigma_1, \Gamma_1' \wedge g, \Delta \vdash \{\Gamma_1'\}\ s_1'\ \{\Gamma_1\}$ (7) - (2) + (5) + hyp.3 + hyp.ind.
- Taking $\Gamma' = \Gamma_1'$ (8)
- $h', \rho' \vDash \Sigma', \Gamma'$ (9) - (6) + (7) + (8)
- By derivation, $g, \Delta \vdash \{\Gamma'\}\ s'\ \{\Gamma_f\}$

$$\frac{g, \Delta \vdash \{\Gamma_1'\}\ s_1'\ \{\Gamma_1\} \qquad g, \Delta \vdash \{\Gamma_1\}\ s_2\ \{\Gamma_2\}}{g, \Delta \vdash \{\Gamma_1'\}\ s_1'; s_2\ \{\Gamma_2\}}$$

(9) - (3) + (6) + (7) + (8)

□

4.3 Soundness - Progress

In this section we prove that the proposed type system satisfies the *progress property* with respect to the small-step operational semantics defined in Section 4.1. Essentially, progress states that if a statement is typable with respect to a given initial store typing environment, then either it is the skip statement or it is always possible to perform another computation step in any state satisfying that initial store typing environment. More formally, if a statement s is typable with respect to a given function g, typing context Δ and store typing environments Γ and Γ_f, i.e. $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma_f\}$. Then, either s is the skip statement or it is always possible to perform another computation step in any state (h, ρ) such that $h, \rho \vDash \Sigma, \Gamma$, for a given heap typing environment Σ. Put formally:

$$(g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma_f\}\ \wedge\ h, \rho \vDash \Sigma, \Gamma) \implies (s = \mathsf{skip}\ \vee\ \exists \Sigma', h', \rho', s' : \langle \Sigma, h, \rho, s\rangle \rightarrow_i \langle \Sigma', h', \rho', s'\rangle) \quad (4.3)$$

Well-Typed Expressions - Progress In order to establish the progress property of our type system, we make use of a new auxiliary property named progress of expression typing. To establish this, we have to prove that if an expression e is given type τ_e by our type system for expressions in a store typing environment Γ and if a store ρ satisfies Γ; then the evaluation of e in ρ yields a value v. Lemma 12 formally establishes this property.

Lemma 12 (Well-typed Expressions - Progress). *Let e be an expression, τ_e a type, ρ a store, Σ a heap typing environment and Γ a store typing environment. Suppose that $\Gamma \vdash e : \tau_e$ and $\rho \vDash_\Sigma \Gamma$. Then $\exists_v :\ [\![e]\!]_\rho = v$.*

Proof. Assume that $\Gamma \vdash e : \tau_e(hyp.1)$ and $\rho \vDash_\Sigma \Gamma(hyp.2)$. Therefore we have that:
Case e is a variable x (hyp.3)

- $\Gamma(x) = \tau_e$ (1) - hyp.1 + hyp.3
- $dom(\rho) = dom(\Gamma)$ (2) - hyp.2
- $x \in dom(\rho)$ (3) - (1) + (2)
- $[\![e]\!]_\rho = \rho(e)$ (4) - (3)
- $\exists_v :\ [\![e]\!]_\rho = v$ (5) - (4)

Case e is a value v' (hyp.3)

- $[\![e]\!]_\rho = v'$ (or $false$) (1) - hyp.3
- $\exists_v :\ [\![e]\!]_\rho = v$ (2) - (1)

Case e is $\underline{+}(e')$ (hyp.4)

- $\exists \tau_e' : \Gamma \vdash e' : \tau_e'$ (1) - hyp.1 + hyp.4
- $\exists v' : [\![e']\!]_\rho = v'$ (2) - (1) + hyp.2 + hyp.ind.
- $\exists v' : [\![\underline{+}(e')]\!]_\rho = \underline{+}(v')$ (3) - (2)
- $\exists v' : [\![e]\!]_\rho = \underline{+}(v')$ (4) - (3)
- $\exists v : [\![e]\!]_\rho = v$ (5) - (4)

Case e is $\underline{+}(e_1, e_2)$ (hyp.3)

- $\exists \tau_{e_1}, \tau_{e_2} : \Gamma \vdash e_1 : \tau_{e_1}$ and $\Gamma \vdash e_2 : \tau_{e_2}$ (1) - hyp.1 + hyp.4
- $\exists v_1, v_2 : [\![e_1]\!]_\rho = v_1$ and $[\![e_2]\!]_\rho = v_2$ (2) - (1) + hyp.2 + hyp.ind.
- $\exists v_1, v_2 : [\![\underline{+}(e_1, e_2)]\!]_\rho = \underline{+}(v_1, v_2)$ (3) - (2)
- $\exists v_1, v_2 : [\![e]\!]_\rho = \underline{+}(v_1, v_2)$ (4) - (3)
- $\exists v : [\![e]\!]_\rho = v$ (5) - (4)

□

Progress Theorem Theorem 6 states that the proposed type system satisfies the Progress property.

Theorem 6 (Progress). *Let g be a function and Δ a typing context. Let h be a heap, ρ a store, and s a statement. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma_f\}$ and $h, \rho \vDash \Sigma, \Gamma$. Then either $s =$ skip or $\exists \Sigma', h', \rho', s' : \langle \Sigma, h, \rho, s \rangle \rightarrow_i \langle \Sigma', h', \rho', s' \rangle$.*

Proof.

Assuming that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma_f\}$ (hyp.1) and $h, \rho \vDash \Sigma, \Gamma$ (hyp.2), we have the following cases:

[SKIP] Suppose that $s = \text{skip}$ (hyp.3). So we are in the $s = \text{skip}$ case.

[ASSIGNMENT] Suppose that $s = x := e$ (hyp.3). We have that:

- $\frac{\Gamma \vdash e:\tau_e\ Closed(\tau_e)}{g, \Delta \vdash \{\Gamma\}\ x:=e\ \{\Gamma[x \mapsto \tau_e]\}}$ is applied (1) - hyp.1 + hyp.3
- $\Gamma \vdash e : \tau_e$ (2) - (1)
- $Closed(\tau_e)$ (3) - (1)
- $\exists_v :\ [\![e]\!]_\rho = v$ (4) - (2) + hyp.2 + Lemma WTE-Progress
- $v \vDash_\Sigma \tau_e$ (5) - (4) + hyp.2 + Lemma WTE-Safety
- $\text{Closed}_\Sigma(v)$ (6) - (2) + (4) + Lemma Closed Values
- $\langle \Sigma, h, \rho, x := e \rangle \rightarrow_i \langle \Sigma, h, \rho[x \mapsto v], \text{skip} \rangle$ (7) - (4) + (6)

[ASSIGNMENT FROM FIELD] Suppose that $s = x := e.f$ (hyp.3). We have that:

- $\frac{\Gamma \vdash e.f:\tau\ Closed(\tau)}{g, \Delta \vdash \{\Gamma\} x:=e.f \{\Gamma[x \mapsto \tau]\}}$ is applied (1) - hyp.1 + hyp.3
- $\Gamma \vdash e.f : \tau$ (2) - (1)
- $Closed(\tau)$ (3) - (1)
- $\exists_l :\ [\![e]\!]_\rho = l$ (4) - (2) + hyp.2 + Lemma WTE-Progress
- $h(l, f) \vDash_\Sigma \tau$ (5) - (2) + (4) + hyp.2 + Lemma WTE-Safety
- $\exists_v : v \vDash_\Sigma \tau$ (6) - (5)
- $\text{Closed}_\Sigma(v)$ (7) - (3) + (6)
- $\langle \Sigma, h, \rho, x := e \rangle \rightarrow_i \langle \Sigma, h, \rho[x \mapsto v], \text{skip} \rangle$ (7) - (2) + (6) + (7)

[FIELD ASSIGNMENT OPEN] Suppose that $s = x.f := e$ (hyp.3) and $\lfloor \Sigma([\![x]\!]_\rho) \rfloor = \circ$ (hyp.4). We have two cases to consider:

- $\frac{\Gamma\vdash e:\tau_e \;\; \Gamma(x)=\{f_i:\tau_i|_{i=1}^{k}\}^\circ,\forall_{i\in\{1,\ldots,k\}} f\neq f_i \;\; \mathsf{Closed}(\tau_e)}{g,\Delta\vdash\{\Gamma\}x.f:=e\{\Gamma[x\mapsto\{f_i:\tau_i|_{i=1}^{k},f:\tau_e\}^\circ]\}}$ or
  $\frac{\Gamma\vdash e:\tau_e \;\; \Gamma(x)=\{f_i:\tau_i|_{i=1}^{k}\}^\circ,\exists_{j\in\{1,\ldots,k\}} f=f_j \;\; \mathsf{Closed}(\tau_e)}{g,\Delta\vdash\{\Gamma\}x.f:=e\{\Gamma[x\mapsto\{f_i:\tau_i|_{i=1,i\neq j}^{k},f:\tau_e\}^\circ]\}}$ is applied.

  (1) - hyp.1 + hyp.3 + hyp.4
- $\exists_l :\; [\![x]\!]_\rho = l$ (2) - (1)
- $\Gamma \vdash e : \tau_e$ (3) - (1)
- $Closed(\tau_e)$ (4) - (1)
- $\exists_v :\; [\![e]\!]_\rho = v$ (5) - (3) + hyp.2 + Lemma WTE-Progress
- $v \vDash_\Sigma \tau_e$ (6) - (3) + (5) + hyp.2 + Lemma WTE-Safety
- $\mathsf{Closed}_\Sigma(v)$ (7) - (4) + (6) + Lemma Closed Values
- $\langle \Sigma, h, \rho, x.f := e\rangle \rightarrow_i \langle \Sigma' = \Sigma[l \mapsto \Sigma(l)[f \mapsto \tau]], h[(l, f) \mapsto v], \rho, \mathsf{skip}\rangle$

  (8) - (1) + (5) + (7) + hyp.4

[FIELD ASSIGNMENT CLOSE] Suppose that $s = x.f := e$ (hyp.3) and $\lfloor\Sigma([\![x]\!]_\rho)\rfloor = \bullet$ (hyp.4). We have that:

- $\frac{\Gamma\vdash e:\tau_e \;\; \Gamma(x)=\{\ldots,f:\tau_f,\ldots\}^\bullet \;\; \tau_e=\tau_f \;\; Closed(\tau_e)}{g,\Delta\vdash\{\Gamma\}x.f:=e\{\Gamma\}}$ is applied

  (1) - hyp.1 + hyp.3 + hyp.4
- $\exists_l :\; [\![x]\!]_\rho = l$ (2) - (1)
- $\Gamma \vdash e : \tau_e$ (3) - (1)
- $Closed(\tau_e)$ (4) - (1)
- $\tau_e = \tau_f$ (5) - (1)
- $\exists_v :\; [\![e]\!]_\rho = v$ (8) - (3) + hyp.2 + Lemma WTE-Progress
- $v \vDash_\Sigma \tau_e$ (9) - (3) + (8) + hyp.2 + Lemma WTE-Safety
- $\mathsf{Closed}_\Sigma(v)$ (10) - (4) + (9) + Lemma Closed Values
- $\Sigma(l, f) = \tau_f = \tau_e$ (11) - (1) + (2) + (5) + hyp.2
- $\Sigma(l, f) = Type_\Sigma(v)$ (12) - (9) + (11)
- $(l, f) \in dom(h)$ (13) - (12) + hyp.2
- $\langle \Sigma, h, \rho, x.f := e\rangle \rightarrow_i \langle \Sigma' = \Sigma[l \mapsto \Sigma(l)[f \mapsto \tau]], h[(l, f) \mapsto v], \rho, \mathsf{skip}\rangle$

  (13) - (2) + (8) + (10) + (12) + (13) + hyp.3 + hyp.4

[FIELD DELETE] Suppose that $s = \mathsf{delete}(x.f)$ (hyp.3). We have that:

- $\frac{\Gamma(x)=\{f_i:\tau_i|_{i=1}^{k}\}^\circ,\exists_{j\in\{1,\ldots,k\}} f=f_j}{g,\Delta\vdash\{\Gamma\}delete\ x.f\{\Gamma[x\mapsto\{f_i:\tau_i|_{i=1,i\neq j}^{k}\}^\circ]\}}$ is applied. (1) - hyp.1 + hyp.3
- $\exists_l :\; [\![x]\!]_\rho = l$ (2) - (1)
- $\lfloor\Gamma(x)\rfloor = \circ$ (3) - (1)
- $\lfloor\Sigma(l)\rfloor = \lfloor\Gamma(x)\rfloor$ (4) - (2) + hyp.2
- $\lfloor\Sigma(l)\rfloor = \circ$ (5) - (1)
- $\langle \Sigma, h, \rho, \mathsf{delete}(x.f)\rangle \rightarrow_i \langle \Sigma[l \mapsto \Sigma(l)^\bullet], h, \rho, \mathsf{skip}\rangle$

  (6) - (2) + (5)

[NEW OBJECT] Suppose that $s = x := \{\}$ (hyp.3). We have that:

- $g, \Delta \vdash \{\Gamma\} x := \{\} \{\Gamma[x \mapsto \{\}^\circ]\}$ is applied. (1) - hyp.1 + hyp.3

[COMMIT] Suppose that $s = commit(x)$ (hyp.3). We have that:

- $\frac{\Gamma(x)=\{f_i:\tau_i|_{i=1}^{k}\}^\circ}{g,\Delta\vdash\{\Gamma\}commit\ x\{\Gamma[x\mapsto\{f_i:\tau_i|_{i=1}^{k}\}^\bullet]}$ is applied. (1) - hyp.1 + hyp.3
- $\exists_l : \ [\![x]\!]_\rho = l$ (2) - (1)
- $\lfloor \Gamma(x) \rfloor = \circ$ (3) - (1)
- $\lfloor \Sigma(l) \rfloor = \lfloor \Gamma(x) \rfloor$ (4) - (2) + hyp.2
- $\lfloor \Sigma(l) \rfloor = \circ$ (5) - (1)
- $\langle \Sigma, h, \rho, commit(x) \rangle \rightarrow_i \langle \Sigma[l \mapsto \Sigma(l)^\bullet], h, \rho, \mathsf{skip} \rangle$

  (6) - (2) + (5)

[IF] Suppose that $s = \mathsf{if}(e)\{s_1\}\ \mathsf{else}\ \{s_2\}$ (hyp.3).

- $\frac{g,\Delta\vdash e:bool\ \ g,\Delta\vdash\{\Gamma_0\}s_1\{\Gamma_1\}\ \ g,\Delta\vdash\{\Gamma_0\}s_2\{\Gamma_2\}}{g,\Delta\vdash\{\Gamma_0\}if(e)\{s_1\}else\{s_2\}\{\Gamma_1\sqcap\Gamma_2\}}$ is applied. (1) - hyp.1 + hyp.3
- $g, \Delta \vdash e : bool$ (2) - (1)

  We have two cases to consider:

  Case $e = true$ (hyp.4)

  - $\langle \Sigma, h, \rho, \mathsf{if}(e)\{s_1\}\ \mathsf{else}\ \{s_2\} \rangle \rightarrow_i \langle \Sigma, h, \rho, s_1 \rangle$ (3.1) - hyp.3 + hyp.4

  Case $e = false$ (hyp.4)

  - $\langle \Sigma, h, \rho, \mathsf{if}(e)\{s_1\}\ \mathsf{else}\ \{s_2\} \rangle \rightarrow_i \langle \Sigma, h, \rho, s_2 \rangle$ (3.2) - hyp.3 + hyp.4

[WHILE] Suppose that $s = \mathsf{while}(e)\{s\}$ (hyp.3). We have that:

- $\langle \Sigma, h, \rho, \mathsf{while}(e)\{s\} \rangle \rightarrow_i \langle \Sigma, h, \rho, \mathsf{if}(e)\{s; \mathsf{while}(e)\{s\}\}\ \mathsf{else}\ \{\mathsf{skip}\} \rangle$ (1) - hyp.3

[SEQUENCING] Suppose that $s = s_1; s_2$ (hyp.3). We have that:

- $\frac{g,\Delta\vdash\{\Gamma\}s_1\{\Gamma_1\}\ \ g,\Delta\vdash\{\Gamma_1\}s_2\{\Gamma_2\}}{g,\Delta\vdash\{\Gamma\}s_1;s_2\{\Gamma_2\}}$ is applied. (1) - hyp.1 + hyp.3
- $g, \Delta \vdash \{\Gamma\}\ s_1\ \{\Gamma_1\}$ (2) - (1)

  We have two cases to consider:

  Case $s_1 \neq \mathsf{skip}$ (hyp.4)

  - $\langle \Sigma, h, \rho, s_1 \rangle \rightarrow_i \langle \Sigma', h', \rho', s_1' \rangle$ (3.1.1) - (2) + hyp.2 + hyp.4 + hyp.ind.
  - $\langle \Sigma, h, \rho, s_1; s_2 \rangle \rightarrow_i \langle \Sigma_1', h_1', \rho_1', s_1'; s_2 \rangle$ (3.1.2) - (3.1.1)

  Case $s_1 = \mathsf{skip}$ (hyp.4)

  - $\langle \Sigma, h, \rho, \mathsf{skip}; s_2 \rangle \rightarrow_i \langle \Sigma, h, \rho, s_2 \rangle$ (3.2.1) - hyp.4 + hyp.2 + hyp.ind.

$\square$

4.4 Function and Return

In this section, we adapt our small-step semantics and soundness proofs to account for function calls and return statements. Subsection 4.4.1 introduces the extended small-step semantics; Subsection 4.4.2 extends the NAOO invariant to call stacks and proves that the extended NAOO invariant is maintained by the semantics; finally, Subsection 4.4.3 concludes with the extended soundness theorems and respective proofs.

4.4.1 Semantics

In order to extend our small-step semantics to take into account function calls, we rely on the notion of *call stack* [16]. Call stacks are used to keep track of the execution context of the calling function. Hence, when evaluating a function call, we extend the current call stack with a record that book-keeps the calling context. Conversely, when evaluating a return statement, the semantics discards the current execution context and recovers the execution context of the calling function (i.e. the function that receives the returned value) from the call stack. Formally, call stacks are generated by the following grammar:

$$cs ::= [\,] \mid (f, x, \rho, s) :: cs \tag{4.4}$$

Where $::$ denotes list concatenation and $[\,]$ the empty list. Essentially, a call stack is a list of 4-tuples of the form (f, x, ρ, s), referred to as call stack records, where: **(1)** f is the identifier of the calling function; **(2)** x is the program variable of the calling function to which the result of the current function is to be assigned; **(3)** ρ is the store of the calling function; and **(4)** s is the continuation of the calling function; that is: the part of the body of the calling function that still remains to be executed once the current function returns. Given a call stack cs, we use the notation $stores(cs)$ to refer to the corresponding list of stores; the function $stores$ is inductively defined as follows:

$$stores(cs) = \begin{cases} [\,] & \text{if } cs = [\,] \\ \rho :: stores(cs') & \text{if } cs = (-, -, \rho, -) :: cs' \end{cases} \tag{4.5}$$

We are now at the position to extend our semantic judgement for statements introduced in Section 4.1 with support for function calls and return statements. To this end, we have to change the format of the semantic judgment to $\langle g, \Sigma, h, \rho, cs, s\rangle \rightarrow_i \langle g', \Sigma', h', \rho', cs', s'\rangle$, with g and cs representing the current function identifier and call stack, and g' and cs' representing the resulting function identifier and call stack.

New Object

$$\frac{l \notin dom(h) \qquad \Sigma' = \Sigma[l \mapsto \{\}^\circ] \qquad h' = h[l \mapsto \{\}]}{\langle g, \Sigma, h, \rho, cs, x := \{\}\rangle \rightarrow_i \langle g, \Sigma', h', \rho[x \mapsto l], cs, \mathsf{skip}\rangle}$$

Function Call

$$\frac{[\![e_i]\!]_\rho = v_i|_{i=1}^n \qquad \mathsf{Closed}_\Sigma(v_i)|_{i=1}^n \qquad \mathsf{body}(f) = s' \qquad \mathsf{params}(f) = x_i|_{i=1}^n \qquad cs' = (g, x, \rho, s) :: cs}{\langle g, \Sigma, h, \rho, cs, x := f(e_1, ..., e_n); s\rangle \rightarrow_i \langle f, \Sigma, h, [x_i \mapsto v_i|_{i=1}^n], cs', s'\rangle}$$

Return

$$\frac{[\![e]\!]_\rho = v \qquad cs = (f, x, \rho', s') :: cs' \qquad \rho'' = \rho'[x \mapsto v]}{\langle g, \Sigma, h, \rho, cs, \mathsf{return}(e)\rangle \rightarrow_i \langle f, \Sigma, h, \rho'', cs', s'\rangle}$$

Top Level Return

$$\frac{[\![e]\!]_\rho = v \qquad cs = [] \qquad \rho'' = \rho[out \mapsto v]}{\langle g, \Sigma, h, \rho, cs, \mathsf{return}(e)\rangle \rightarrow_i \langle g, \Sigma, h, \rho'', cs, \mathsf{skip}\rangle}$$

Figure 4.2: Small-Step semantics for statements - function call: $\langle g, \Sigma, h, \rho, cs, s\rangle \rightarrow_i \langle g', \Sigma', h', \rho', cs', s'\rangle$

Figure 4.2 gives a selection of the extended semantic rules. In particular, it includes the rules that handle function calls, the return statement, and the new object. Note that the rules introduced in Figure 4.1 can be straightforwardly adapted to this setting as their current function and call stack do not change. For instance, the assignment from field rule becomes:

Assignment From Field

$$\frac{[\![e]\!]_\rho = l \qquad h(l, f) = v \qquad \mathsf{Closed}_\Sigma(v)}{\langle g, \Sigma, h, \rho, x := e.f, cs\rangle \rightarrow_i \langle g, \Sigma, h, \rho[x \mapsto v], \mathsf{skip}, cs\rangle}$$

The rules presented in Figure 4.2 are explained below.

[New Object] This rule is applied when assigning a new object to a variable x. The rule finds a location l which is not in the domain of the heap and adds such location to both heap and heap typing environment, mapping it to an empty object and to an empty open object type, respectively. The rule also maps x to l in the store and proceeds to a skip statement.

[Function Call] This rule is applied when the first element of a sequence statement is a function call with closed arguments. The rule starts by evaluating the supplied arguments, obtaining a list of closed values $v_i|_{i=1}^n$. Then, the rule obtains the list of formal parameters of the function f to be executed $x_i|_{i=1}^n$ and the statement s' corresponding to the body of the function. Then a new store is created, mapping the formal parameters of the function to the supplied arguments, $[x_i \mapsto v_i|_{i=1}^n]$. Afterwards, the rule appends the current function, g, the variable x, the starting store, ρ and the second element of the sequence statement, s, to the call stack. Finally, the evaluation proceeds to s' with current function f.

[Return] This rule is applied when a return statement is found and the call stack is not empty. The rule simply evaluates the expression being returned obtaining a value v. Then, it pops the call stack obtaining (f, x, ρ', s'). The transition continues using f as current function, $\rho[x \mapsto v]$ as store, s' as statement and with the popped call stack, cs'.

[Top Level Return] This rule is applied when a return statement is found and the call stack is empty. The rule finds the value v to be returned from evaluating the expression e and simply asssigns it to a special variable out in the store. The rule proceeds to a skip statement.

4.4.2 Semantic Properties

Call Stack Satisfiability Just as for the heap and store, we introduce here the notion of call stack satisfiability. Defined inductively, a call stack with (f, x, ρ, s) at its top is said to satisfy a function g, a typing context Δ and a heap typing environment Σ if there exists two store typing environments Γ and Γ' such that s is typable under f, Δ, Γ and Γ', and for any value v which satisfies the return type of g, we have that $\rho[x \mapsto v] \vDash_\Sigma \Gamma$. Moreover, the popped call stack also needs to satisfy f, Δ and Γ. The empty call stack always satisfies any g, Δ, Σ.

Definition 10 (Call Stack satisfiability)**.** *Given a heap typing environment Σ, a typing context Δ and a function g, a call stack cs is said to satisfy g, Δ, Σ, written $cs \vDash g, \Delta, \Sigma$, if:*

- $cs = []$ *or*
- $cs = (f, x, \rho, s) :: cs'$ *such that:*
  - $\exists_{\Gamma,\Gamma'} \forall_v \; v \vDash_\Sigma \Delta_r(g) \Rightarrow \rho[x \mapsto v] \vDash_\Sigma \Gamma \wedge f, \Delta \vdash \{\Gamma\} \; s \; \{\Gamma'\}$
  - $cs' \vDash f, \Delta, \Sigma$

Where $\Delta_r(g)$ denotes the return type of g.

Essentially, the call stack satisfiability property guarantees that all the continuations in the current call stack are typable and that all stores in the current call satisfy the corresponding store typing environment if the extended with a value of the appropriate type.

No Aliasing for Call Stack (NACS) In order to deal with aliasing and mutation, our type system enforces the NAOO property, meaning that only closed objects can be referenced by more than one pointer. With the addition of the call stack, we have to extend this invariant to take into account the stores that form the call stack. To this end, we introduce the notion of no aliasing for call stack (NACS).

Definition 11 (No Aliasing for Call Stack)**.** *Let cs be a call stack, ρ_n a store, and Σ a heap typing environment; cs and ρ_n satisfy the no aliasing property with respect to Σ, written $NACS(\Sigma, \rho_n, cs)$, if:*

- stores$(cs) = [\rho_0, ..., \rho_{n-1}]$
- $\forall_{l \in Locs} \forall_{x,y \in Vars} \forall_{i,j} \; \rho_i(x) = \rho_j(y) = l \wedge i \neq j \Rightarrow \lfloor \Sigma(l) \rfloor = \bullet$

Essentially, the NACS property means that only closed objects may be referenced by variables pertaining to different stores. In other words, if two variables x and y in different stores ρ_i and ρ_j reference the same location l, then the object pointed to by l must be closed.

The proposed type system enforces the NACS invariant. However, such as with the NAOO property, we do not prove that the type system does enforce the NACS invariant directly. Instead, we instrumented the operational semantics so that it also enforces the NACS invariant and will later prove that typable programs cannot be rejected by the semantics for violating the NACS invariant. Lemma 13 proves that the operational semantics preserve the NACS invariant. In the given proof we consider only the most significant cases which are the return, functional call, new object and field assignment open.

Lemma 13 (NACS Preservation)**.** *Let g and f be functions, Σ and Σ' heap typing environments, Γ and Γ' store typing environments, h and h' heaps, ρ and ρ' stores, s and s' statements, and cs and cs' call stacks. Suppose that $\langle g, \Sigma, h, \rho, cs, s\rangle \rightarrow_i \langle f, \Sigma', h', \rho', cs', s'\rangle$, $h, \rho \vDash \Sigma, \Gamma$ and $NACS(\Sigma, \rho, cs)$, then $NACS(\Sigma', \rho', cs')$.*

Proof. The proof follows by induction on the step s.

Therefore, assuming that $\langle g, \Sigma, h, \rho, cs, s\rangle \rightarrow_i \langle f, \Sigma', h', \rho', cs', s'\rangle$ (hyp.1), $h, \rho \vDash \Sigma, \Gamma$ (hyp.2) and (p, cs) satisfies $NACS$ with respect to Σ(hyp.3) we have the following base cases:

[RETURN] Suppose that $s = \mathsf{return}(e)$ (hyp.4). We have that:

- $\mathsf{stores}(cs') = (\rho_0, ..., \rho_{n-2}), \Sigma' = \Sigma, \rho' = \rho_{n-1}$ (1) - hyp.1 + hyp.4
- $NACS(\Sigma', \rho', cs')$ (2) - (1) + hyp.3

[FUNCTION CALL] Suppose that $s = x := f(e_1, ..., e_n); s$ (hyp.4). We have that:

- $\rho_n = \rho, \mathsf{stores}(cs') = (\rho_0, ..., \rho_{n-1}, \rho), \Sigma' = \Sigma, \rho' = [x_i \mapsto [\![e_i]\!]_\rho|_{i=1}^n]$ (1) - hyp.1 + hyp.4
- $\mathsf{Closed}_\Sigma([\![e_i]\!]_\rho)|_{i=1}^n$ (2) - hyp.1 + hyp.4
- $\lfloor \Sigma([\![e_i]\!]_{\rho'}) \rfloor = \bullet$ for $i \in \{1, ..., n\}$ (3) - (2)
- $NACS(\Sigma', \rho', cs')$ (4) - (3) + hyp.3

[NEW OBJECT] Suppose that $s = x := \{\}$ (hyp.4). We have that:

- $\mathsf{stores}(cs') = \mathsf{stores}(cs), \Sigma' = \Sigma[l \mapsto \{\}^\circ], \rho' = \rho[x \mapsto l], l \notin dom(h)$ (1) - hyp.1 + hyp.4

  Renaming ρ' as ρ_n, we have to prove that:

$$\forall_{\hat{l},x,y,i,j}\ \rho_i(x) = \rho_j(y) = \hat{l} \wedge i \neq j \Rightarrow \lfloor \Sigma'(\hat{l}) \rfloor = \bullet$$

- $dom(h) = dom(\Sigma)$ (2) - hyp.2
- $\forall_{k \in \{0,...,n-1\}} \forall_z\ \rho_k(z) \neq l$ (3) - (1) + Lemma No Segmentation Fault

  Therefore there are two cases to consider:

  Case $\hat{l} \in dom(h)$ (hyp.6):

  – $\hat{l} \neq l$ (4.1.1) - hyp.6

  – $\hat{l} = \rho(x) = \rho'(x) = \rho_n(x)$ (4.1.2) - (1) + (4.1.1) + hyp.5

  – $\forall_{\hat{l},x,y,i,j}\ \rho_i(x) = \rho_j(y) = \hat{l} \wedge i \neq j \Rightarrow \lfloor \Sigma'(\hat{l}) \rfloor = \bullet$ (4.1.3) - (1) + (4.1.2) + hyp.3

  Case $\hat{l} \notin dom(h)$ (hyp.6):

  – $\hat{l} = l$ (4.2.1) - (1) + (2) + hyp.6

  – $\forall_{x,y} \neg \exists_{i,j \in \{1,...,n\}, i \neq j} : \rho_i(x) = \rho_j(y) = \hat{l}$ (4.2.2) - (3) + (4.2.1)

  – $\forall_{\hat{l},x,y,i,j}\ \rho_i(x) = \rho_j(y) = \hat{l} \wedge i \neq j \Rightarrow \lfloor \Sigma'(\hat{l}) \rfloor = \bullet$ (4.2.3) - (4.2.2)

- $NACS(\Sigma', \rho', cs')$ (5) - (4.1.3) + (4.2.3)

[FIELD ASSIGNMENT OPEN] Suppose that $s = x.f := e$ (hyp.4) and $\lfloor \Sigma([\![x]\!]_\rho) \rfloor = \circ$ (hyp.5). We have that:

- $\mathsf{stores}(cs') = \mathsf{stores}(cs), \Sigma' = \Sigma[[\![x]\!]_\rho \mapsto \Sigma(l)[f \mapsto Type_\Sigma([\![e]\!]_\rho)]], \rho' = \rho$

(1) - hyp.1 + hyp.4 + hyp.5

- If $\lfloor\Sigma(\hat{l})\rfloor = \bullet$, $\Sigma(\hat{l}) = \Sigma'(\hat{l})$ (2) - (1) + hyp.5
- $\forall_{\hat{l},x,y,i,j}\ \rho_i(x) = \rho_j(y) = \hat{l} \wedge i \neq j \Rightarrow \lfloor\Sigma'(\hat{l})\rfloor = \bullet$ (3) - (2) + hyp.3
- $NACS(\Sigma', \rho', cs')$ (4) - (1) + (3)

□

In the presented proof we refer to the fact that there is no segmentation fault, meaning that no store, including the ones in the call stack, have as an image locations which are not part of the heap. Formally:

Definition 12 (No Segmentation Fault). *Let h be a heap, cs a call stack with* $\mathsf{stores}(cs) = (\rho_0, ..., \rho_{n-1})$ *and ρ_n a store. It is said that there is no segmentation fault if:*

- $\forall_{l \in Loc} \forall_{k \in \{0,...,n\}}\ \rho_k(z) = l \Rightarrow l \in dom(h)$

Despite not proving this fact directly, our semantics does, in fact, not have segmentation fault as the store only starts pointing to locations which are already part of the heap. Moreover, our language does not have garbage collection. When an object is created it stays in the heap forever as the delete command deletes object fields, not objects themselves.

4.4.3 Soundness

We now prove that our full type system satisfies the Progress and Preservation [17] properties with respect to the extended operational semantics.

Soundness - Preservation With the introduction of the call stack we have to consider an updated version of our Preservation theorem. Analogously to the type safety theorem for our big-step semantics presented in Section 3.5, with the introduction of function calls, we now also require that the global typing context Δ types the program p as hypothesis.

Theorem 7 (Soundness - Preservation). *Let p be a program containing functions g and f, Δ a typing context, Σ and Σ' heap typing environments, Γ and Γ' store typing environments. Let h and h' be heaps, ρ and ρ' stores and s and s' statements. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$, $\langle g, \Sigma, h, \rho, cs, s\rangle \to_i \langle f, \Sigma', h', \rho', cs', s'\rangle$, $h, \rho \vDash \Sigma, \Gamma$, $cs \vDash g, \Delta, \Sigma$ and $\Delta \vdash p$. Then there exist two typing environments $\hat{\Gamma}$ and $\hat{\Gamma}'$ such that the following are true: $h', \rho' \vDash \Sigma', \hat{\Gamma}$; $f, \Delta \vdash \{\hat{\Gamma}\}\ s'\ \{\hat{\Gamma}'\}$; and $cs' \vDash f, \Delta, \Sigma'$.*

Proof. The proof follows by induction on the derivation of:

$$\langle g, \Sigma, h, \rho, cs, s\rangle \to_i \langle f, \Sigma', h', \rho', cs', s'\rangle$$

Hence, assuming that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$ (hyp.1), $\langle g, \Sigma, h, \rho, cs, s\rangle \to_i \langle f, \Sigma', h', \rho', cs', s'\rangle$ (hyp.2), $h, \rho \vDash \Sigma, \Gamma$ (hyp.3), $cs \vDash g, \Delta, \Sigma$ (hyp.4) and $\Delta \vdash p$ (hyp.5), we have the following base cases:

[RETURN] Suppose that $s = \mathsf{return}(e)$ (hyp.6). We have that:

- $g, \Delta \vdash \{\Gamma\}\ \mathsf{return}(e)\ \{\Gamma\}$ is applied. (1) - hyp.1 + hyp.6
- $\langle g, \Sigma, h, \rho, cs, \mathsf{return}(e)\rangle \to_i \langle f, \Sigma', h', \rho', cs', s'\rangle$ (2) - hyp.2 + hyp.6

  We now have two case to consider:

  Case $cs = []$ (hyp.7)

  - $\Sigma' = \Sigma, h' = h, \rho' = \rho$ (3.1.1) - (1) + hyp.7
  - $\hat{\Gamma} = \Gamma$ (3.1.2) - (2)
  - $h', \rho' \vDash \Sigma', \hat{\Gamma}$ (3.1.3) - (3.1.1) + (3.1.2)
  - $cs' \vDash f, \Delta, \Sigma$ (3.1.4) - hyp.7

  Case $cs \neq []$ (hyp.7)

  - $cs = (f, x, \rho'', s') :: cs'$ (3.2.1) - (2) + hyp.7
  - $[\![e]\!]_\rho = v$ (3.2.2) - (2) + hyp.7
  - $\Sigma' = \Sigma, h' = h, \rho' = \rho''[x \mapsto v]$ (3.2.3) - (2) + hyp.7
  - $\Gamma \vdash e : \Delta_r(g)$ (3.2.4) - (1)
  - $h' \vDash \Sigma'$ (3.2.5) - (3.2.3) + hyp.3
  - $\exists_{\hat{\Gamma}, \hat{\Gamma}'} \forall_{v'}\ v' \vDash_\Sigma \Delta_r(g) \Rightarrow \rho''[x \mapsto v'] \vDash_\Sigma \hat{\Gamma} \wedge f, \Delta \vdash \{\hat{\Gamma}\}\ s'\ \{\hat{\Gamma}'\}$ (3.2.6) - (3.2.1) + hyp4
  - $v \vDash_\Sigma \Delta_r(g)$ (3.2.7) - (3.2.2) + (3.2.4) + hyp.3 + Lemma WTE-Safety
  - $\rho''[x \mapsto v] \vDash_\Sigma \hat{\Gamma} \wedge f, \Delta \vdash \{\hat{\Gamma}\}\ s'\ \{\hat{\Gamma}'\}$ (3.2.8) - (3.2.6) + (3.2.7)
  - $\rho' \vDash_\Sigma \hat{\Gamma} \wedge f, \Delta \vdash \{\hat{\Gamma}\}\ s'\ \{\hat{\Gamma}'\}$ (3.2.9) - (3.2.3) + (3.2.8)
  - $h', \rho' \vDash \Sigma', \hat{\Gamma}$ (3.2.10) - (3.2.5) + (3.2.9)
  - $cs' \vDash f, \Delta, \Sigma$ (3.2.11) - (3.2.1) + hyp.4

[FUNCTION CALL] Suppose that $s = x := f(e_1, ..., e_n); \hat{s}$ (hyp.6). We have that:

- $g, \Delta \vdash \{\Gamma\}\ x := f(e_1, ..., e_n); \hat{s}\ \{\Gamma'\}$ is applied. (1) - hyp.1 + hyp.5
- $g, \Delta \vdash \{\Gamma\}\ x := f(e_1, ..., e_n)\ \{\Gamma[x \mapsto \Delta_r(f)]\}$ is applied. (2) - (1)
- $g, \Delta \vdash \{\Gamma[x \mapsto \Delta_r(f)]\}\ \hat{s}\ \{\Gamma'\}$ (3) - (1)
- $\langle g, \Sigma, h, \rho, cs, x := f(e_1, ..., e_n); \hat{s}\rangle \to_i \langle f, \Sigma', h', \rho', cs', s'\rangle$ (4) - hyp.2 + hyp.5
- $cs' = (g, x, \rho, \hat{s}) :: cs$ (5) - (4)
- $[\![e_i]\!]_\rho = v_i|_{i=1}^n$ (6) - (4)
- $\mathsf{body}(f) = s'$ (7) - (4)
- $\mathsf{params}(f) = x_i|_{i=1}^n$ (8) - (4)
- $\Sigma' = \Sigma, h' = h, \rho' = [x_i \mapsto v_i|_{i=1}^n]$ (9) - (4) + (6) + (8)
- $\Gamma \vdash e_i : \Delta_i(f)|_{i=1}^n$ (10) - (2)
- $v_i \vDash_\Sigma \Delta_i(f)|_{i=1}^n$ (11) - (6) + (10) + hyp.3 + Lemma WTE-Safety
- Taking $\hat{\Gamma} = [x_i \mapsto \Delta_i(f)|_{i=1}^n]$ we have that $\rho' \vDash_{\Sigma'} \hat{\Gamma}$ (12) - (9) + (11)
- $h' \vDash \Sigma'$ (13) - (9) + hyp.3

- $h', \rho' \vDash \Sigma', \hat{\Gamma}$ (14) - (12) + (13)
- $\exists_{\hat{\Gamma}'} : f, \Delta \vdash \{\hat{\Gamma}\}\ s'\ \{\hat{\Gamma}'\}$ (15) - (7) + hyp.6
- $cs' \vDash f, \Delta, \Sigma$ (16) - (5) + (16.1) + (16.2) + hyp.4

  Given that cs' is equal to cs extended with the stack frame $(g, x, \rho, \hat{s})$, we have to prove that the continuation in the new stack frame is typable; more concretely, we have to prove that when assigning a value v' of the appropriate type to x, the continuation $\hat{s}$ is typable under two variable typing environments, $\underline{\Gamma}$ and $\underline{\Gamma}'$. Put formally:

$$\exists_{\underline{\Gamma}, \underline{\Gamma}'} \forall_{v'}\ v' \vDash_\Sigma \Delta_r(f) \Rightarrow \rho[x \mapsto v'] \vDash_\Sigma \underline{\Gamma} \wedge g, \Delta \vdash \{\underline{\Gamma}\}\ \hat{s}\ \{\underline{\Gamma}'\}$$

  Taking $\underline{\Gamma} = \Gamma[x \mapsto \Delta_r(f)]$, $\underline{\Gamma}' = \Gamma'$ and assuming that $v' \vDash_\Sigma \Delta_r(f)$ (hyp.7) we have that:

  - $\rho[x \mapsto v'] \vDash_\Sigma \underline{\Gamma}$ (16.1) - hyp.3 + hyp.7
  - $g, \Delta \vdash \{\underline{\Gamma}\}\ \hat{s}\ \{\underline{\Gamma}'\}$ (16.2) - (3)

The remaining cases are essentially similar to what is presented in Theorem 5 □

We use $\langle g, \Sigma, h, \rho, cs, s\rangle \to_i^j \langle f, \Sigma', h', \rho', cs', s'\rangle$ to denote a j-step transition. When $j = 1$ we simply write $\langle g, \Sigma, h, \rho, cs, s\rangle \to_i \langle f, \Sigma', h', \rho', cs', s'\rangle$ just as before. Thus, we leverage induction to extend Theorem 7 to its multi-step version given below.

Corollary 1 (N-step Transition Soundness - Preservation)**.** *Let p be a program containing functions g and f, Δ a typing context, Σ and Σ' heap typing environments, Γ and Γ' store typing environments. Let h and h' be heaps, ρ and ρ' stores and s and s' statements. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$, $\langle g, \Sigma, h, \rho, cs, s\rangle \to_i^n \langle f, \Sigma', h', \rho', cs', s'\rangle$, for any $n \in \mathbb{N}$, $h, \rho \vDash \Sigma, \Gamma$, $cs \vDash g, \Delta, \Sigma$ and $\Delta \vdash p$. Then there exist two typing environments $\hat{\Gamma}$ and $\hat{\Gamma}'$ such that the following are true: $h', \rho' \vDash \Sigma', \hat{\Gamma}$; $f, \Delta \vdash \{\hat{\Gamma}\}\ s'\ \{\hat{\Gamma}'\}$; and $cs' \vDash f, \Delta, \Sigma'$.*

Proof. The proof follows by induction on n. Assuming that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$ (hyp.1), $\langle g, \Sigma, h, \rho, cs, s\rangle \to_i^n \langle f, \Sigma', h', \rho', cs', s'\rangle$ (hyp.2), $h, \rho \vDash \Sigma, \Gamma$ (hyp.3), $cs \vDash g, \Delta, \Sigma$ (hyp.4) and $\Delta \vdash p$ (hyp.5), we have the following:

Case $n = 0$: It follows that $\langle g, \Sigma, h, \rho, cs, s\rangle \to_i^0 \langle f, \Sigma', h', \rho', cs', s'\rangle$. In such case, we have that: $f = g, \Sigma' = \Sigma, h' = h, \rho' = \rho, cs' = cs, s' = s$, therefore, by hypothesis, $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$ (hyp.1), $cs \vDash g, \Delta, \Sigma$ (hyp.4) and $h, \rho \vDash \Sigma, \Gamma$ (hyp.3).

Case $n = k + 1$: Note that $\langle g, \Sigma, h, \rho, cs, s\rangle \to_i^{k+1} \langle f, \Sigma', h', \rho', cs', s'\rangle$ is equivalent to $\langle g, \Sigma, h, \rho, cs, s\rangle \to_i^k \langle \underline{g}, \underline{\Sigma}, \underline{h}, \underline{\rho}, \underline{cs}, \underline{s}\rangle$ (1) and $\langle \underline{g}, \underline{\Sigma}, \underline{h}, \underline{\rho}, \underline{cs}, \underline{s}\rangle \to_i \langle f, \Sigma', h', \rho', cs', s'\rangle$ (2), for some $\underline{g}, \underline{\Sigma}, \underline{h}, \underline{\rho}, \underline{cs}$ and $\underline{s}$. Applying the induction hypothesis to hyp.1, (1), hyp.3, hyp.4, and hyp.5, we conclude that there exist $\underline{\hat{\Gamma}}$ and $\underline{\hat{\Gamma}'}$ such that:

$$\underline{h}, \underline{\rho} \vDash \underline{\Sigma}, \underline{\hat{\Gamma}}\ \text{(3)} \qquad \underline{g}, \underline{\Delta} \vdash \{\underline{\hat{\Gamma}}\}\ \underline{s}\ \{\underline{\hat{\Gamma}'}\}\ \text{(4)} \qquad \underline{cs} \vDash \underline{g}, \underline{\Delta}, \underline{\Sigma}\ \text{(5)}$$

Applying Theorem 7 to (4), (2), (3), (5), and hyp.5, we conclude that there exist two typing environments $\hat{\Gamma}$ and $\hat{\Gamma}'$ such that the following are true:

$$h', \rho' \vDash \Sigma', \hat{\Gamma} \qquad f, \Delta \vdash \{\hat{\Gamma}\}\ s'\ \{\hat{\Gamma}'\} \qquad cs' \vDash f, \Delta, \Sigma'$$

□

Soundness - Progress We finish our small-step semantics chapter by extending the Progress theorem to the updated transition rules. As in Theorem 7, some hypothesis have to be added. We now require that the global typing context Δ types the program p and that the existing call stack satisfies the current function g, Δ, and the heap typing environment Σ'. Formally:

Theorem 8 (Progress)**.** *Let g be a function and Δ a typing context. Let h be a heap, ρ a store, and s a statement. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma_f\}$, $h, \rho \vDash \Sigma, \Gamma$, $cs \vDash g, \Delta, \Sigma$ and $\Delta \vdash p$. Then either $s = \mathsf{skip}$ or $\exists f, \Sigma', h', \rho', cs', s' : \langle g, \Sigma, h, \rho, cs, s\rangle \rightarrow_i \langle f, \Sigma', h', \rho', cs', s'\rangle$.*

Proof. Assuming that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma_f\}$ (hyp.1) and $h, \rho \vDash \Sigma, \Gamma$ (hyp.2), $cs \vDash g, \Delta, \Sigma$ (hyp.3) and $\Delta \vdash p$ (hyp.4) we have the following cases:

[Return] Suppose that $s = \mathsf{return}(e)$ (hyp.5). We have that:

- $\frac{\Gamma(g)=(\tau_1',...,\tau_n')\rightarrow\tau_e\ \Gamma\vdash e:\tau_e}{g,\Delta\vdash\{\Gamma\}return\ e\{\Gamma\}}$ is applied
  (1) - hyp.1 + hyp.5
- $\Gamma \vdash e : \tau_e$ (2) - (1)
- $\exists_v :\ [\![e]\!]_\rho = v$ (3) - (2) + hyp.2 + Lemma WTE-Progress

  There are two cases to consider:

  Case $cs = []$ (hyp.6)
- $\langle g, \Sigma, h, \rho, cs, \mathsf{return}(e)\rangle \rightarrow_i \langle g, \Sigma, h, \rho, cs, \mathsf{skip}\rangle$ (4.1.1) - hyp.6

  Case $cs \neq []$ (hyp.6)
- $\langle g, \Sigma, h, \rho, (f, x, \rho', s') :: cs', \mathsf{return}(e)\rangle \rightarrow_i \langle f, \Sigma, h, \rho'[x \mapsto v], cs', s'\rangle$
  (4.2.1) - (3) + hyp.6

[Function Call] Suppose that $s = x := f(e_1, ..., e_n); \hat{s}$ (hyp.5). We have that:

- $\frac{g,\Delta\vdash\{\Gamma_0\}s_1\{\Gamma[x\mapsto\tau]\}\ g,\Delta\vdash\{\Gamma[x\mapsto\tau]\}s_2\{\Gamma_2\}}{g,\Delta\vdash\{\Gamma_0\}s_1;s_2\{\Gamma_2\}}$ is applied. (1) - hyp.1 + hyp.5
- $\frac{\Gamma\vdash e_i:\tau_i|_{i=1}^n\ \Delta(f)=(\tau_1,...,\tau_n)\rightarrow\tau\ Closed(\tau_i)|_{i=1}^n}{g,\Delta\vdash\{\Gamma\}x:=f(e_i|_{i=1}^n)\{\Gamma[x\mapsto\tau]\}}$ (2) - (1)
- $\Gamma \vdash e_i : \tau_i|_{i=1}^n$ (3) - (2)
- $\Delta(f) = (\tau_1, ..., \tau_n) \rightarrow \tau$ (4) - (2)
- $Closed(\tau_i)|_{i=1}^n$ (5) - (2)
- $\exists_{v_i} :\ [\![e_i]\!]_\rho = v_i|_{i=1}^n$ (6) - (3) + hyp.2 + Lemma WTE-Progress
- $v_i \vDash_\Sigma \tau_i|_{i=1}^n$ (7) - (3) + (6) + hyp.2 + Lemma WTE-Safety
- $\langle g, \Sigma, h, \rho, cs, x := f(e_1, ..., e_n); s\rangle \rightarrow_i \langle f, \Sigma, h, [x_i \mapsto v_i|_{i=1}^n], (g, x, \rho, s) :: cs, \mathsf{body}(f)\rangle$
  (6) - (5)

The remaining cases are essentially similar to what is presented in Theorem 6. □

Chapter 5

Related Work

The research literature covers a wide variety of program analysis techniques for JavaScript, such as: type systems [18, 19], abstract interpreters [20], points-to analyses [21], program logics [14, 22], operational semantics [23–25], just to mention a few. We focus our analysis of the related work on type systems for JavaScript-like languages.

Thiemann [9] was the first to propose a type system for a subset of JavaScript. While this type system considered some of the dynamic aspects of JavaScript, such as extensible objects, dynamic function calls, and type coercions, it also ignored various important aspects of the language, most notably JavaScript's prototype-based inheritance mechanism. Importantly, the type system proposed by Thiemann is flow-insensitive, meaning that variables and object fields are not allowed to change type over time. To overcome this issue, Anderson et al. [26] later proposed a type system that allows JavaScript objects to evolve in a controlled manner. The key idea behind this work is to classify object fields as potential or definite; potential fields can have their types change while definite fields cannot. The idea of potential/definite fields is reminiscent of our open/closed objects. These two strategies have, however, different trade-offs with both being able to typecheck legal programs that are rejected by the other.

Later, Jensen et al. [27] proposed the first sound type analysis for real JavaScript code, called TAJS. The proposed analysis is flow-sensitive, allowing the types of variables and object fields to change over time, and based on abstract interpretation [28]. The main contribution of this analysis is the design of a complex lattice to reason about unary and binary operations in JavaScript, which takes into account JavaScript's implicit type coercions.

The TypeScript programming language [8] was designed with the goal of adding optional types to JavaScript, taking opportunity of JavaScript's flexibility, while at the same time providing some of the advantages of statically typed languages, such as informative compiling errors and automatic code completion. Client-side JavaScript programs make extensive use of external APIs that are not available for static typing, thus the analysis of TypeScript programs requires the specification of interface declarations for the external libraries that a program may use. However, interface declarations are humanly written and not necessarily by the authors of the libraries, therefore leaving room for errors that can compromise the soundness of the typing process. To solve this program, Feldthaus et al. [29] proposed a method for checking the correction of TypeScript declaration files with respect to JavaScript library implementations.

More recently, some type systems were developed with the objective of enabling efficient ahead-of-time compilation for JavaScript. The first purposed work with this objective was from Choit et al. [30] which supports prototype-based inheritance, structural subtyping, and method updates. Later on, Chandra et al. [10] expanded on this work and incorporated additional annotations, thus enabling the type system to better differentiate between readable and writable object fields.

In the rest of the chapter, we provide a detailed discussion of the papers that are closest to our work and which guided the design of our type system. The papers are presented in chronological order.

Towards a Type System for Analyzing JavaScript Programs

Thiemann [9] presents the first published sound type system for a subset of JavaScript. This type system supports singleton types, subtyping, and first-class methods, and guarantees that:

- the first sub-expression of a function call or a new expression always evaluates to a function;
- there are no null pointer dereferences, i.e. when accessing a field of an object, the expression that denotes the object does not evaluate to the null object or the value undefined;
- arithmetic operators are not applied to objects (unless the object is a wrapper for a number);
- there are no implicit type coercions, meaning that primitive types are not implicitly converted to object types.

Importantly, this type system does not support prototype inheritance, making it considerably simpler than its successors.

Object types, written as $Obj(\tau)(x_1 : \tau_1, ..., x_n : \tau_n)(\tau')$, include a *row type*, $(x_1 : \tau_1, ..., x_n : \tau_n)$, mapping the fields of the object to their corresponding types and a *default type* τ' denoting the type of all fields not mentioned explicitly in the row type. Additionally, object types include a special *feature type* τ, indicating if the object is a wrapper for a primitive value or a function. If that is not the case, the feature type τ is set to $undefined$. For instance, the object $\{x : 2,\ y : "foo"\}$ has type $Obj(undefined)(x : number,\ y : string)(undefined)$. Note that we set the default type to $undefined$ to indicate that objects with this type only have the fields x and y; hence, the result of inspecting any other field, say z, is the value $undefined$.

To better illustrate the usage of feature types, consider the example below:

```
var a = new String("black hole")
a.x = 51
a.x //OK
```

```
var b = "black hole"
b.x = 51
b.x //BAD
```

In the second example, JavaScript creates a string b, and then tries to assign a value to its field x. Since b has type $string$ it cannot have fields, thus JavaScript creates a wrapper object around b and assigns the value 51 to its field x. Despite this, the newly created wrapper object will be garbage-collected immediately after, as b is still bound to the primitive string *"black hole"* instead of the newly created wrapper object. Since b is still a primitive string, it does not contain the field x; hence, the inspection of this field yields the value $undefined$. The same behavior does not happen in the first example, which explicitly creates a string object in the first line. Hence, the assignment of value 51 to field x works as usual, with no need for creating a wrapper object, and the following field look up also yields the value 51. The type system proposed by Thiemann prevents implicit coercions, disallowing the second example. Below we will explain how.

In the first example, the string object created in the first line has type:

$$Obj(String)(x : number)(undefined)$$

Where the feature type is $String$ and the row type contains the field x wit type $number$. Hence, the assignment in the second line is allowed to go through, given that the value 51 has type $number$. In contrast, the type system blocks the field assignment of the second example, $b.x = 51$, because b has type $string$, instead of being an object type.

Towards Type Inference for JavaScript

The authors of [26] present a formalism for an object oriented language JS_0, based on JavaScript. The authors define an operational semantics, a type inference algorithm and a sound static type system for JS_0, making use of structural types. The proposed type system supports: recursive types, subtyping, and allows objects to evolve in a controlled manner by marking their fields as potential or definite. This field marking strategy uses a similar idea to our open/closed objects: each object's field is marked as potential or definite, and, by tracking this, the object may see some evolution. This evolution is only possible for potential fields, which become definite when they are assigned a value. This constitutes yet another approach to handling one of the major difficulties of typing mutable objects: when we update an object field with a value of a different type the type of the object changes.

To better understand these ideas consider the following example from the paper:

```
function Date(x):(t1 * number -> t2) {
    this.mSec = x;  this.add = addFn; this;
}
function addFn(x):(t2 * t2 -> t2) {
    this.mSec = this.mSec + x.mSec;  this;
}
//Main
t2 x = new Date(1000);
t2 y = new Date(100);
x.add(y);
```

Where $t_1 = [mSec : (number, \circ),\ add : ((t_2 \times t_2 \to t_2), \circ)]$ and $t_2 = \mu\ \alpha.[mSec : (number, \bullet),\ add : ((\alpha \times \alpha \to \alpha), \bullet)]$. As annotated, the function $Date$ has type $t_1 \times number \to t_2$, meaning that: (1) the receiver object has type t_1 (recall that the receiver object is the object bound to the keyword $this$); (2) the function receives an argument of type $number$; (3) and the return value has type t_2. In turn, the function $addFn$ has type $t_2 \times t_2 \to t_2$, meaning that its receiver object, argument and return value all have type t_2. In t_1 the associated fields are still marked as potential ($\circ$) but in t_2, as they get assigned inside the function $Date$, they become definite ($\bullet$). Once a field becomes definite, its type can no longer be changed. Furthermore, the type system forbids accesses to potential fields, meaning that, one can only access a field once it becomes definite.

Despite the similarities between this type system and the one we present, there are some considerable differences. First, we close (i.e. make definite) objects instead of object fields, which requires less annotations and streamlines the tracking process. Furthermore, this difference in approaches leads to differences in expressivity; for example, the type system proposed in [26] does not account for field deletion, while our type system allows for field deletion as long as the corresponding object is still open.

Understanding TypeScript

Designed for the development of large applications, TypeScript is a strongly typed programming language that extends JavaScript by adding optional static typing to it. As such, every JavaScript program is also a TypeScript program. The authors of [8] aim to capture the essence of TypeScript by providing a precise definition for its associated type system based on a core subset of TypeScript referred to as Featherweight TypeScript (**FTS**). More concretely, the authors consider two subsets of TypeScript: a safe subset, denoted as **safeFTS**, that is proven to be sound, and an unsafe subset, **prodFTS**, which is closer to the full TypeScript language and is shown to be unsound (even when annotations are abundant).

The key difference between TypeScript and the previously presented type systems is the fact that TypeScript is not aimed at soundness. This choice allows it to be applicable to the vast majority of JavaScript libraries, which would be difficult to type using traditional type systems.

The key features of TypeScript are the following:

- Full Erasure: TypeScript is converted to JavaScript, therefore its types leave no trace in the JavaScript emitted by the TypeScript's transpiler;
- Structural Types: as objects are often built from scratch (and not from classes) in JavaScript structural types are adopted instead of nominal;
- Unified Object Types: objects, functions, constructors and arrays are all treated as object types;
- Type Inference: type inference is used to minimize the number of type annotations needed to be provided explicitly;
- Gradual Typing: parts of the program are statically typed, and others are dynamically typed through the usage of the special type any;
- Downcasting: expressions can be converted from a parent object to a child object when compatible. Usually this is compiled to a dynamic check, which is not possible here as TypeScript is converted to JavaScript;
- Covariance: object subtyping is covariant in the types of the object's fields. For instance, if $\tau_1 \leq \tau_2$, then $\{f : \tau_1\} \leq \{f : \tau_2\}$. This type of subtyping is known to be unsound as we demonstrate below.

Below, we expand on the unsoundness problems introduced by covariant subtyping of object types. Consider the example below consisting of two interfaces $Person$ and $Vegetarian$.

```
interface Person {
    eats: any,
    name: string,
    eat (food : any) :boolean
}
```

```
interface Vegetarian {
    eats: Vegetable,
    name: string,
    eat (food : any) :boolean
}
```

As TypeScript uses covariant subtyping for object types, it considers the type $Vegetarian$ to be a subtype of the type $Person$, since: $Vegetable \leq any$, $string \leq string$, and $(any \Rightarrow boolean) \leq (any \Rightarrow boolean)$. However, this allows us to break the internal invariants of the class $Vegetarian$ as illustrated by the following example:

```
1  // Maria is a Vegetarian
2  var p : Person = Maria;
3  p.eats = Meat.beef;
4  p.eat (Meat.beef);
```

Given that $Vegetarian$ is a subtype of $Person$, the assignment of line 2 typechecks. Hence, the program is allowed to assign $Maria$ to variable p of type $Person$. Next, it sets the field $eats$ of p (in this case, $Maria$) to $Food.Meat$ and, then, it feeds meat to $Maria$, clearly breaking the invariant of the $Vegetarian$ type.

Safe & Efficient Gradual Typing for TypeScript

The authors of [31] tackle the lack of soundness present in current solutions to add gradual typing to JavaScript by proposing a new and practical gradual type system, prototyped for expediency as a ”Safe” compilation mode for TypeScript that enforces stricter static checks and embeds residual runtime checks in compiled code. Its implementation is fully integrated as a branch of the TypeScript-0.9.5 compiler, which programmers can opt in by providing a flag to the compiler.

Safe TypeScript is composed by two phases. The first phase is the standard TypeScript compilation process whilst the second phase confirms, soundly, the types inferred before. If none of the phases find a static error, Safe TypeScript rewrites the program and instruments objects with runtime type information (RTTI) and checks. This process makes use of two key differentiators:

- Differential subtyping: a form of coercive subtyping that computes the minimum amount of runtime type information that must be added to each object. Essentially, as tagging an object may be costly, only dynamically used objects are tagged, and even for those only the parts used in the dynamic type discipline require tagging.

- Partial Erasure: used to safely and selectively erase type information. Essentially in Safe TypeScript, any characterizes only values associated with runtime type information, thus the values that are not associated with RTTI - erased types - are not subtypes of any.

To better understand these ideas consider the following snippet of Safe TypeScript code (left) and its compiler translation to JavaScript (right) from the paper:

```
interface Point { x : number, y : number}
interface Circle { center:Point;
↪  radius:number }
function copy(p:Point, q:Point) { q.x=p.x;
↪  q.y=p.y; }
function f(q:any) {
    var c = q.center;
    copy(c, {x:0, y:0});
    q.center = {x:"bad"}; }
function g(circ:Circle) : number {
    f(circ);
return circ.center.x; }
```

```
RT.reg("Point",{"x":RT.num,"y":RT.num});
RT.reg("Circle",{"center":RT.mkRTTI("Point"),
↪  "radius":RT.num});
function copy(p, q) { q.x=p.x; q.y=p.y; }
function f(q) {
    var c = RT.readField(q,"center");
    copy(RT.checkAndTag(c,
    ↪  RT.mkRTTI("Point")),{x:0,y:0});
    RT.writeField(q, "center", {x:"bad"});
    ↪  }
function g(circ) {
    f(RT.shallowTag(circ,
    ↪  RT.mkRTTI("Circle")));
return circ.center.x; }
```

In the example above, two types are defined ($Point$ and $Circle$) and three functions ($copy$, g and f). Notice that function g provides f with a $Circle$ type, whilst f receives any. The function g is supposed to return a $number$, however, $circ.center$ is no longer a $Point$, since $q.center$ is of another type, and thus its field x is a $string$ instead of a $number$. This is a type error, but Safe TypeScript cannot detected this error statically (nor simple TypeScript), thus the detection is made at runtime. In order to detect this, Safe TypeScript first transforms interface definitions to calls to RT and follows by using RTTI to express invariants that must be enforced at runtime. For example, $circ$ must be a $Circle$ and this is expressed by instrumenting $circ$ with the function $RT.shadowTag$ which maintains RTTI information. These invariants are propagated throughout the program by the $readField$ function, which tags RTTI information to objects. In this case it indicates that $q.center$ must be a $Point$. Afterwards, the function $checkAndTag$ is used to check whether the provided information has the required type, in this case, it is used to check if c is a supertype for $Point$, which it is and thus this information is passed as $RTTI$. However, the call $writeField(o, f, v)$ is used to check whether the value v being written to the field f of object o is consistent with the existing typing invariants, which is not the case here as $\{x : bad\}$ cannot be typed as a $Point$.

SJS: a Typed Subset of JavaScript with Fixed Object Layout

The authors of [30] present a static type system for a subset of JavaScript that guarantees that objects have a statically-known layout at allocation time. This type system is used as part of an ahead-of-time (AOT) compiler for JavaScript that generates efficient code which can run in standard Javascript engines. This type system supports prototype-based inheritance, structural subtyping, and method updates. In order to achieve this, the type system uses the three following key ideas:

- Each object is annoted with a row type that maps its fields to their respective types. The row type reflects the entire prototype chain of an object, meaning that if an object can access a field a with a given value then the row type of the object must include the field a with its respective type.
- Object types are additionally annotated with a set of *owned fields* and a set of *inheritor-owned fields*. The first correspond to the fields directly defined in the object as opposed to somewhere in

its prototype chain. The second correspond to the fields that must also be owned by the inheritors of the object (the objects that have the current object in their prototype chains). The idea of owned fields is that an object can only have a field updated if that field is defined in the object itself. However, the methods defined in an object can be invoked on both that object or on any of its inheritors. Hence, to guarantee that methods can only change local fields, all fields that might be updated by the methods of an object must be included in its set of inheritor owned fields. This enforces that the set of inheritor owned fields is a subset of the set of owned fields.

- The authors distinguish between precise types, which are used when the dynamic type of an object is known statically, and approximate types, which are used when it is not. An approximate object type is a supertype of the precise object type with the same fields and the same owned fields. However, approximated types do not have inheritor owned fields as they cannot be used as prototypes. The subtyping relation ensures that a precise type can be cast into an abstract type.

To better understand these ideas consider the following examples from the paper:

```
1  var o1 = { a : 1, f : function (x) { this.a = 2 } }
2  var o2 = { b : 1, __proto__ : o1 }
3  o1.a = 3 //OK
4  o2.a = 2 //BAD
5  o2.f() //BAD
```

In the example above $o1$ has type $o1 : \{a : number, f : number \Rightarrow void\}^{P(\{a,f\},\{a\})}$, meaning that a and f are its owned fields and a is its only inheritor owned field; this means that all the inheritors of $o1$, such as $o2$, must own at least the field a in order to update it via the method f. Since $o2$ has type $o2 : \{a : number, b : number, f : number \Rightarrow void\}^{P(\{b\},\{\})}$, the considered type system blocks the assignment in line two because $o2$ does not include the field a, which is the single inheritor owned field of $o1$ in order to exclude erroneous executions such as the one illustrated in the example.

Let us now take a closer look at the usage of approximate types. Consider the following example:

```
1  var o5 = { a : 1, b : 2, f : function (x) { this.a = 2 } }
2  var o6 = { a : 1, b : 3, f : function (x) { this.b = 3 } }
3  o6.f = function (x) { this.b = 4 } // OK
4  var o7 = expr ? o5 : o6
5  o7.f = function (x) { this.b = 4 } // BAD
6  console.log(o7.a); // OK
```

In this example $o5$ and $o6$ both have precise types - $o5 : \{a : number, b : number, f : number \Rightarrow void\}^{P(\{a,b,f\},\{a\})}$ and $o6 : \{a : number, b : number, f : number \Rightarrow void\}^{P(\{a,b,f\},\{b\})}$. As $o7$ is the result of a conditional statement involving $o5$ and $o6$, its type will be the least upper bound between the types of $o5$ and $o6$: $\{a : number, b : number, f : number \Rightarrow void\}^{A(\{a,b,f\})}$. This least upper bound stems from the fact that the dynamic type of $o7$ is not known statically, therefore it will have an approximate type with a set of fields equals to the intersection of owned fields from $o5$ and $o6$. Thus the method update in line 5 is forbidden, as one cannot update a method field of an object with an abstract type. In contrast, the method update in line 3 is allowed to go through, as the object $o6$ has a precise type. Despite this, the fields of $o7$ can still be read.

Type Inference for Static Compilation of JavaScript

The authors of [10] present a sound type system and inference algorithm for a subset of JavaScript that uses lower and upper bound propagation to infer types and discover type errors. In the following we focus on the type system as the type inference algorithm is out of the scope of this thesis.

Analogously to the type system of the paper presented in the previous section, this type system is also used to enable an optimizing ahead-of-time (AOT) compiler for JavaScript. It supports prototype-based inheritance, structural subtyping, recursive types, and first-class methods, and was made to enforce the following two properties:

- **Type Compatibility** - different types cannot be assigned to the same variable;
- **Access Safety** - fields that are neither available locally nor in the prototype chain cannot be read; and fields that are not locally available cannot be written.

In order to satisfy both of those properties, objects are annotated with four row types: O^r, the fields that can be read, O^w, the fields that can be written, O^{mr}, the fields that may be read by methods attached to the object and O^{mw}, the fields that may be written by methods attached to the object. The first two are as expected: O^r is the set of fields available locally or in the prototype chain and O^w is the set of fields available locally. The last two contain the fields that might be read or written through the use of methods. Moreover object types are annotated with three possible qualifiers: a prototypal qualifier, $P(mr, mw)$, a non-prototypal concrete qualifier, NC, and a non-prototypal abstract qualifier, NA. Only objects with prototypal qualifiers can be used as prototypes or have their methods updated, but have to be converted to non-prototypal concrete first to enable subtyping. The distinction between the non-prototypal qualifiers, NC and NA, is that objects with concrete qualifiers may have their methods invoked on them, while objects with abstract qualifiers may not. Object types are written as follows: $O : \{\langle O^r\rangle | \langle O^w\rangle\}^{P(\langle O^{mr}\rangle, \langle O^{mw}\rangle)}$ if prototypal, or simply $O : \{\langle O^r\rangle | \langle O^w\rangle\}^{N*}$ if not.

To better understand these ideas, let us consider the following example from the paper:

```
var v1 = { d : 1, // o1
    m : function (x) { this.a = x + this.d }}
var v2 = { a : 2 } proto v1; // o2
v2.m(3); //OK
v2.m("foo"); //BAD
var v3 = { b : 4 } proto v2; // o3
v3.m(4); //BAD
```

For clarity, each object in the example is annotated with a unique identifier. Below, we explain the types assigned by the type system to $o1$, $o2$, and $o3$:

- The object $o1$ has type $\{\langle d : number, m : number \Rightarrow void\rangle | \langle d, m\rangle\}^{P(\langle d,a\rangle, \langle a\rangle)}$. As $o1$ has no prototype, its readable fields $\langle d, m\rangle$ coincide with its writable fields. As m is the only method attached to $o1$, the method-readable fields of $o1$ will be those read by m and the method-writable fields of $o1$ will be those written to by m. Hence, $o1$ has the qualifier $P(\langle d, a\rangle, \langle a\rangle)$.

- The object $o2$ has type $\{\langle d : number, m : number \Rightarrow void, a : number\rangle | \langle a\rangle\}^{P(\langle d,a\rangle,\langle a\rangle)}$. This object has a single writable field, a, corresponding to the single field that it directly defines. Moreover, as its prototype is $o1$, its readable fields include the readable fields of $o1$, d, and m, besides its own field a. As m is, by inheritance, the only method attached to $o2$, the method-readable fields and method-writable fields of $o2$ coincide with those of $o1$.

- The object $o3$ has type $\{\langle d : number, m : number \Rightarrow void, a : number, b : number\rangle | \langle b\rangle\}^{P(\langle d,a\rangle,\langle a\rangle)}$. This object has a single writable field, b, corresponding to the single field that it directly defines. Moreover, as its prototype is $o2$, its readable fields include the readable fields of $o2$, d, m, and a, besides its own field b. As m is, by inheritance, the only method attached to $o3$, the method-readable fields and method-writable fields of $o3$ coincide with those of $o2$. Hence, $o3.m(4)$ cannot be invoked as the set of writable fields does not contain the set of method-writable fields.

Coming back to the topic of type qualifiers, we can see that all these three objects have the same prototypal qualifier, $P(\langle d, a\rangle, \langle a\rangle)$. However, only the type of $o2$ may be converted to a non-prototypal concrete type (NC), since it is the only one whose sets of method-read and -write fields are contained in the corresponding sets of read and write fields.

Chapter 6

Conclusions

The ECMA-SL project was created with the goal of assisting with the analysis of JavaScript programs by first compiling the programs to be analysed to the ECMA-SL language. To this end, the ECMA-SL project includes a compilation tool chain from JavaScript to ECMA-SL, which has at its core a reference interpreter of the 5th version of the ECMAScript standard called ECMARef5. As the ECMAScript standard is now at its 12th version, in order for the ECMA-SL project to be relevant for the analysis of current JavaScript programs, its compilation pipeline must be adapted to the more recent versions of the ECMAScript standard. To achieve this, one must first adapt the ECMARef5 interpreter that sits at its core. However, such extension is rendered extremely difficult due to the fact that ECMA-SL is an untyped language.

This thesis contributes to the overall ECMA-SL project by designing Typed ECMA-SL, a typed version of ECMA-SL, together with a sound type system for checking Typed ECMA-SL programs. We believe that the implementation of the proposed system would make ECMA-SL substantially easier to use by statically detecting a variety of programming errors that would be, otherwise, only detected dynamically via testing.

In summary, the contributions of this thesis are the following:

Typed ECMA-SL The first contribution of this thesis is the design of Typed ECMA-SL, a typed extension of ECMA-SL [7]. Whilst doing this extension we aimed to minimize the number of extra annotations required, thus enabling already proficient developers in ECMA-SL to easily transition to Typed ECMA-SL.

Type System The second contribution of this thesis is the design of a type system for checking Typed ECMA-SL programs. The purposed type system is both flow-sensitive and sound. Flow-sensitivity allows for program variables and objects to change their types during execution. Soundness ensures that well-typed programs cannot go wrong. Soundness is particularly difficult to achieve in the setting of ECMA-SL due to the dynamicity of the language, which includes extensible objects and the dynamic deletion of object fields.

Soundness Proofs The third contribution of this thesis is the development of two soundness proofs for the proposed type system: one based on a big-step operational semantics and another one based on a small-step operational semantics. The two proofs enabled us to better understand the trade-offs between both types of semantics when proving the soundness of a type system. On the one hand, the small-step operational semantics is more involved than its big-step version, requiring the definition of call stacks and an additional invariant for constraining the ways in which call stacks can be manipulated. On the other hand, the big-step semantics has to model erroneous cases explicitly as they cannot be otherwise differentiated from non-terminating derivations.

6.1 Future Work

Implementing the type system The clear next step is to implement the proposed type system. With the implementation of the proposed type system, the ECMARef interpreter can then be extended and adapted to newer versions of the JavaScript standard.

Enabling Closed Objects to Become Open When a closed object is assigned to one and only one variable, changing its type has a local impact, therefore it might as well be treated as an open object. Thus, by introducing a more fine-grained control mechanism over object aliasing, we could potentially re-open an object after it being closed. For instance, we envisage the introduction of an uncommit command for opening an object after closing it. This would require further instrumenting the syntax of types to book-keep the pointers to values of a specific type.

Extending the type system In the near future we would like to extend ECMA-SL with support for recursive types, union types and subtyping, as these features would be useful for the development of a typed version of the ECMARef interpreter. For instance, recursive types are essential to model object types with a recursive structure, such as abstract syntax trees.

Bibliography

[1] JavaScript Across the World Wide Web. `https://w3techs.com/technologies/details/cp-javascript/`. Accessed: 2021-10-25.

[2] Node.js Increasing Usage. `https://w3techs.com/technologies/details/ws-nodejs`, . Accessed: 2021-10-25.

[3] Node.js. `https://nodejs.org/en/`, . Accessed: 2021-10-25.

[4] F. Quinaz. Precise information flow control for javascript. 2021.

[5] ECMAScript® Language Specification, 5.1 Edition / June 2011. `https://262.ecma-international.org/ecma-262/5.1/ECMA-262.pdf`, . Accessed: 2021-10-30.

[6] ECMAScript® Language Specification, 12 Edition / June 2021. `https://www.ecma-international.org/wp-content/uploads/ECMA-262_12th_edition_june_2021.pdf`, . Accessed: 2021-10-30.

[7] L. Loureiro. Ecma-sl - a platform for specifying and running the ecmascript standard. 2021.

[8] G. Bierman, M. Abadi, and M. Torgersen. Understanding typescript. In R. Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44202-9.

[9] P. Thiemann. Towards a type system for analyzing javascript programs. In M. Sagiv, editor, *Programming Languages and Systems*, pages 408–422, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31987-0.

[10] S. Chandra, C. S. Gordon, J.-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, and Y. Choi. Type Inference for Static Compilation of JavaScript (Extended Version). *arXiv e-prints*, art. arXiv:1608.07261, Aug. 2016.

[11] G. Kahn. Natural semantics. In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '87, page 22–39, Berlin, Heidelberg, 1987. Springer-Verlag. ISBN 354017219X.

[12] G. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 07 2004. doi: 10.1016/j.jlap.2004.05.001.

[13] https://github.com/tc39/test262. `https://github.com/tc39/test262`. Accessed: 2021-10-30.

[14] J. F. Santos, P. Gardner, P. Maksimović, and D. Naudžiūnienė. Towards logic-based verification of javascript programs. In L. de Moura, editor, *Automated Deduction – CADE 26*, pages 8–25, Cham, 2017. Springer International Publishing. ISBN 978-3-319-63046-5.

[15] J. Fragoso Santos, P. Maksimović, D. Naudžiūnienundefined, T. Wood, and P. Gardner. Javert: Javascript verification toolchain. *Proc. ACM Program. Lang.*, 2(POPL), Dec. 2017. doi: 10.1145/3158138. URL `https://doi.org/10.1145/3158138`.

[16] J. Fragoso Santos, P. Maksimović, S.-E. Ayoun, and P. Gardner. Gillian, part i: A multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 927–942, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386014. URL `https://doi.org/10.1145/3385412.3386014`.

[17] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *INFORMATION AND COMPUTATION*, 115:38–94, 1992.

[18] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In J. Palsberg and Z. Su, editors, *Static Analysis*, pages 238–255, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03237-0.

[19] A. Chaudhuri, P. Vekris, S. Goldman, M. Roch, and G. Levi. Fast and precise type checking for javascript, 2017.

[20] K. Dewey, V. Kashyap, and B. Hardekopf. A parallel abstract interpreter for javascript. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, page 34–45, USA, 2015. IEEE Computer Society. ISBN 9781479981618.

[21] D. Jang and K.-M. Choe. Points-to analysis for javascript. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, page 1930–1937, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605581668. doi: 10.1145/1529282.1529711. URL `https://doi.org/10.1145/1529282.1529711`.

[22] P. A. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for javascript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 31–44, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310833. doi: 10.1145/2103656.2103663. URL `https://doi.org/10.1145/2103656.2103663`.

[23] M. Bodin, A. Charguéraud, D. Filaretti, S. Maffeis, A. Schmitt, and G. Smith. A trusted mechanised javascript specification. In *In Proc. POPL*, 2014.

[24] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for javascript. In G. Ramalingam, editor, *Programming Languages and Systems*, pages 307–325, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-89330-1.

[25] D. Park, A. Stefănescu, and G. Roşu. Kjs: A complete formal semantics of javascript. *ACM SIGPLAN Notices*, 50:346–356, 06 2015. doi: 10.1145/2813885.2737991.

[26] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In A. P. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, pages 428–452, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31725-8.

[27] S. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. volume 5673, pages 238–255, 01 2009. ISBN 978-3-642-03236-3. doi: 10.1007/978-3-642-03237-0_17.

[28] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery. ISBN 9781450373500. doi: 10.1145/512950.512973. URL `https://doi.org/10.1145/512950.512973`.

[29] A. Feldthaus and A. Møller. Checking correctness of typescript interfaces for javascript libraries. In *In Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA*, pages 1–16. ACM, 2014.

[30] W. Choi, S. Chandra, G. C. Necula, and K. Sen. SJS: A type system for javascript with fixed object layout. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, volume 9291 of *Lecture Notes in Computer Science*, pages 181–198. Springer, 2015.

[31] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & efficient gradual typing for typescript. In *POPL '15 Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 167–180, January 2015. ISBN 978-1-4503-3300-9. URL `https://www.microsoft.com/en-us/research/publication/safe-efficient-gradual-typing-typescript/`.

Appendix A

Satisfiability Preservation Lemmas

A.1 Field Update

Lemma 14 (Heap Update). *Let h and h' be heaps, Σ and Σ' heap typing environments, ρ a store, v a value, l a location and τ a type. Suppose that $h \vDash \Sigma$, $h' = h[l \mapsto h(l)[f \mapsto v]]$, $\Sigma' = \Sigma[l \mapsto \Sigma(l)[f \mapsto \tau]]$, $NAOO_1(h, \Sigma)$,* Closed$_\Sigma(v)$*, $\lfloor \Sigma(l) \rfloor = \circ$ and $v \vDash_\Sigma \tau$. Then $h' \vDash \Sigma'$, .*

Proof. Assume that $h \vDash \Sigma$ (hyp.1), $h' = h[l \mapsto h(l)[f \mapsto v]]$ (hyp.2), $\Sigma' = \Sigma[l \mapsto \Sigma(l)[f \mapsto \tau]]$ (hyp.3), $NAOO_1(h, \Sigma)$ (hyp.4), Closed$_\Sigma(v)$ (hyp.5), $\lfloor \Sigma(l) \rfloor = \circ$ (hyp.6) and $v \vDash_\Sigma \tau$ (hyp.7).

Notice that $h' \vDash \Sigma'$ is equivalent to provent all the following conditions:

1. $dom(h') = dom(\Sigma')$
2. $\forall_{l \in dom(h')} dom(h'(l)) = dom(\Sigma'(l))$
3. $\forall_{l \in dom(h')} \forall_{f \in dom(h'(l))} h'(l, f) \vDash_{\Sigma'} \Sigma'(l, f)$

(1) It comes by:

- $dom(h') = dom(h)$ and $dom(\Sigma') = dom(\Sigma)$ (2.1.1) - hyp.2 + hyp.3
- $dom(h) = dom(\Sigma)$ (2.1.2) - hyp.1
- $dom(h') = dom(\Sigma')$ (2.1.3) - (2.1.1) + (2.1.2)

(2) For the second point we have two cases to consider:

Case $\hat{l} = l$:

- $dom(h'(\hat{l})) = dom(h(\hat{l})) \backslash \{f\}$ and $dom(\Sigma'(\hat{l})) = dom(\Sigma(\hat{l})) \backslash \{f\}$
  (2.1.1) - hyp.2 + hyp.3
- $dom(h(\hat{l})) = dom(\Sigma(\hat{l}))$ (2.1.2) - hyp.1
- $dom(h'(\hat{l})) = dom(\Sigma'(\hat{l}))$ (2.1.3) - (2.1.1) + (2.1.2)

Case $\hat{l} \neq l$

- $dom(h'(\hat{l})) = dom(h(\hat{l}))$ and $dom(\Sigma'(\hat{l})) = dom(\Sigma(\hat{l}))$ (2.1) - hyp.2 + hyp.3
- $dom(h(\hat{l})) = dom(\Sigma(\hat{l}))$ (2.2) - hyp.1

- $dom(h'(\hat{l})) = dom(\Sigma'(\hat{l}))$ (2.3) - (2.1) + (2.2)

(3) We have again two cases to consider:

Case $(\hat{l}, \hat{f}) = (l, f)$

- $h'(\hat{l}, \hat{f}) = v$ (3.1.1) - hyp.2
- $\Sigma'(\hat{l}, \hat{f}) = \tau$ (3.1.2) - hyp.3

Let us now consider the following two subcases:

- τ is a primitive type
  - $v \vDash_{\Sigma'} \tau$ (3.1.3.1.1) - hyp.7
- τ is not a primitive type
  - If $v \neq l$ then $\Sigma'(v) = \Sigma(v) = \tau$, thus $v \vDash_{\Sigma'} \tau$ (3.1.3.2.1) - hyp.3 + hyp.7
  - If $v = l$ then $\lfloor \Sigma(v) \rfloor = \bullet$, contradiction (3.1.3.2.2) - hyp.5 + hyp.6
  - $v \vDash_{\Sigma'} \tau$ (3.1.3.2.3) - (3.1.3.2.1) + (3.1.3.2.2)
- $v \vDash_{\Sigma'} \tau$ (3.1.3) - (3.1.3.1.1) + (3.1.3.1.3)
- $h'(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma'(\hat{l}, \hat{f})$ (3.1.4) - (3.1.1) + (3.1.2) + (3.1.3)

Case $(\hat{l}, \hat{f}) \neq (l, f)$

- $h'(\hat{l}, \hat{f}) = h(\hat{l}, \hat{f})$ (3.2.1) - hyp.2
- $\Sigma'(\hat{l}, \hat{f}) = \Sigma(\hat{l}, \hat{f})$ (3.2.2) - hyp.3

Thus, considering two subcases:

- $h(\hat{l}, \hat{f})$ is a value of primitive type
  - $h(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma(\hat{l}, \hat{f})$ (3.2.3.1.1) - hyp.17
- $h(\hat{l}, \hat{f})$ is not a value of primitive type
  - If $h(\hat{l}, \hat{f}) \neq l$ then $\Sigma'(h(\hat{l}, \hat{f})) = \Sigma(h(\hat{l}, \hat{f}))$, thus $h(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma(\hat{l}, \hat{f})$
    (3.2.3.2.1) - hyp.3 + hyp.1
  - If $h(\hat{l}, \hat{f}) = l$ then $\lfloor \Sigma(h(\hat{l}, \hat{f})) \rfloor = \circ$, contradiction
    (3.2.3.2.2) - hyp.4 + hyp.6
  - $h(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma(\hat{l}, \hat{f})$ (3.2.3.2.3) - (3.2.3.2.1) + (3.2.3.2.2)
- $h(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma(\hat{l}, \hat{f})$ (3.2.3) - (3.2.3.1.1) + (3.2.3.1.3)
- $h'(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma'(\hat{l}, \hat{f})$ (3.2.4) - (3.2.1) + (3.2.2) + (3.2.3)

$\square$

Lemma 15 (Store Update)**.** *Let Σ and Σ' be heap typing environments, Γ and Γ' store typing environments, ρ a store, x a variable, f a field and τ a type. Suppose that $\rho \vDash_\Sigma \Gamma$, $\Sigma' = \Sigma[\rho(x) \mapsto \Sigma(\rho(x))[f \mapsto \tau]]$, $\Gamma' = \Gamma[x \mapsto \Gamma(x)[f \mapsto \tau]]$, $\lfloor \Sigma(\rho(x)) \rfloor = \circ$ and $NAOO_2(\rho, \Sigma)$. Then $\rho \vDash_{\Sigma'} \Gamma'$.*

Proof. Assume that $\rho \vDash_\Sigma \Gamma(hyp.1)$, $\Sigma' = \Sigma[\rho(x) \mapsto \Sigma(\rho(x))[f \mapsto \tau]](hyp.2)$, $\Gamma' = \Gamma[x \mapsto \Gamma(x)[f \mapsto \tau]](hyp.3)$, $\lfloor\Sigma(\rho(x))\rfloor = \circ(hyp.4)$ and $NAOO_2(\rho, \Sigma)(hyp.5)$

We want to prove that $\forall_{y \in dom(\Gamma')}\ \rho(y) \vDash_{\Sigma'} \Gamma'(y)$. Let us consider two cases:

Case $y = x$

- $\Gamma'(y) = \Gamma'(x), \rho(y) = \rho(x)$ (1)
- $\Sigma(\rho(x)) = \Gamma(x)$ (2) - hyp.1
- $\Sigma'(\rho(x)) = \Gamma'(x)$ (3) - (2) + hyp.2 + hyp.3
- $\rho(x) \vDash_{\Sigma'} \Gamma'(x)$ (4) - (3)
- $\rho(y) \vDash_{\Sigma'} \Gamma'(y)$ (5) - (1) + (4)

Case $y \neq x$

- $\Gamma'(y) = \Gamma(y)$ (1) - hyp.3

  Considering two subcases:
- $\rho(y)$ is a value of primitive type
  - $\rho(y) \vDash_\Sigma \Gamma(y)$ (2.1.1) - hyp.1
  - $\rho(y) \vDash_{\Sigma'} \Gamma'(y)$ (2.1.2) - (1) + (2.1.1)
- $\rho(y)$ is not a value of primitive type:
  - $\rho(y) \neq \rho(x)$ (2.2.1) - hyp.4 + hyp.5
  - $\Sigma'(\rho(y)) = \Sigma(\rho(y))$ (2.2.2) - (2.2.1) + hyp.4
  - $\rho(y) \vDash_\Sigma \Gamma(y)$ (2.2.3) - hyp.1
  - $\Sigma(\rho(y)) = \Gamma(y)$ (2.2.4) - (2.2.3)
  - $\Sigma'(\rho(y)) = \Gamma(y)$ (2.2.5) - (2.2.2) + (2.2.4)
  - $\rho(y) \vDash_{\Sigma'} \Gamma(y)$ (2.2.6) - (2.2.3) + (2.2.5)
  - $\rho(y) \vDash_{\Sigma'} \Gamma'(y)$ (2.1.2) - (1) + (2.2.6)

□

Lemma 16 (Heap Update - Type Unchanged)**.** *Let h be an heap, l a location, f a field, τ a type and Σ a heap typing environment. Suppose that $h(l, f) \vDash_\Sigma \tau$, $v' \vDash_\Sigma \tau$, $h' = h[(l, f) \mapsto v']$ and $h \vDash \Sigma$. Then $h' \vDash \Sigma$.*

Proof. Assume that $h(l, f) \vDash_\Sigma \tau$ (hyp.1), $v' \vDash_\Sigma \tau$ (hyp.2), $h' = h[(l, f) \mapsto v']$ (hyp.3) and $h \vDash \Sigma$(hyp.4).
We start by noticing that $h' \vDash \Sigma \Leftrightarrow \forall_{\hat{l} \in dom(\Sigma)}\ h'(\hat{l}) \vDash_\Sigma \Sigma(\hat{l})$.
Then we have to prove that $\forall_{\hat{l} \in dom(\Sigma)}\ h'(\hat{l}) \vDash_\Sigma \Sigma(\hat{l})$. Choosing any $\hat{l}$ there are two cases to consider: $\hat{l} = l$ and $\hat{l} \neq l$

Case $\hat{l} = l$ (hyp.5). Admit that, without lose of generality, $h(l) = \{f_i : v_i|_{i=1}^n, f : v\}^*$

- $h'(\hat{l}) = \{f_i : v_i|_{i=1}^n, f : v'\}^*$ (1) - hyp.3 + hyp.5
- $\Sigma(\hat{l}) = \{f_i : \tau_i|_{i=1}^n, f : \tau\}^*$ (2) - hyp.1 + hyp.4 + hyp.5

- $\forall_{1 \leq i \leq n} v_i \vDash_\Sigma \tau_i$ (3) - hyp.4
- $h'(\hat{l}) \vDash_\Sigma \Sigma(\hat{l})$ (4) - hyp.2 + (1) + (2) + (3)

Case $\hat{l} \neq l (hyp.5)$

- $h(\hat{l}) = h'(\hat{l})$ (1) - hyp.3
- $h'(\hat{l}) \vDash_\Sigma \Sigma(\hat{l})$ (2) - hyp.4 + (1)

□

A.2 Field Delete

Lemma 17 (Heap Delete). *Let h and h' be heaps, Σ and Σ' heap typing environments, ρ a store, l a location and f a field. Suppose that $h \vDash \Sigma$, $h' = h\backslash(l, f)$, $\Sigma' = \Sigma\backslash(l, f)$ and $NAOO_1(h, \Sigma)$, $\lfloor \Sigma(l) \rfloor = \circ$. Then $h' \vDash \Sigma'$.*

Proof. Assume that $h \vDash \Sigma$ (hyp.1), $h' = h\backslash(l, f)$ (hyp.2), $\Sigma' = \Sigma\backslash(l, f)$ (hyp.3), $NAOO_1(h, \Sigma)$ (hyp.4) and $\lfloor \Sigma(l) \rfloor = \circ$ (hyp.6).

Notice that $h' \vDash \Sigma'$ is equivalent to provent all the following conditions:

1. $dom(h') = dom(\Sigma')$
2. $\forall_{l \in dom(h')} dom(h'(l)) = dom(\Sigma'(l))$
3. $\forall_{l \in dom(h')} \forall_{f \in dom(h'(l))} h'(l, f) \vDash_{\Sigma'} \Sigma'(l, f)$

(1) It comes by:

- $dom(h') = dom(h)$ and $dom(\Sigma') = dom(\Sigma)$ (1.1.1) - hyp.2 + hyp.3
- $dom(h) = dom(\Sigma)$ (1.1.2) - hyp.1
- $dom(h') = dom(\Sigma')$ (1.1.3) - (1.1.1) + (1.1.2)

(2) For the second point we have two cases to consider:

Case $\hat{l} = l$:

- $dom(h'(\hat{l})) = dom(h(\hat{l}))\backslash\{f\}$ and $dom(\Sigma'(\hat{l})) = dom(\Sigma(\hat{l}))\backslash\{f\}$

  (2.1.1) - hyp.2 + hyp.3
- $dom(h(\hat{l})) = dom(\Sigma(\hat{l}))$ (2.1.2) - hyp.1
- $dom(h'(\hat{l})) = dom(\Sigma'(\hat{l}))$ (2.1.3) - (2.1.1) + (2.1.2)

Case $\hat{l} \neq l$

- $dom(h'(\hat{l})) = dom(h(\hat{l}))$ and $dom(\Sigma'(\hat{l})) = dom(\Sigma(\hat{l}))$ (2.1) - hyp.2 + hyp.3
- $dom(h(\hat{l})) = dom(\Sigma(\hat{l}))$ (2.2) - hyp.1
- $dom(h'(\hat{l})) = dom(\Sigma'(\hat{l}))$ (2.3) - (2.1) + (2.2)

(3) Since (l, f) was deleted we only have one case to consider:

Case $(\hat{l}, \hat{f}) \neq (l, f)$

- $h'(\hat{l}, \hat{f}) = h(\hat{l}, \hat{f})$ (3.2.1) - hyp.2
- $\Sigma'(\hat{l}, \hat{f}) = \Sigma(\hat{l}, \hat{f})$ (3.2.2) - hyp.3

Thus, considering two subcases:

- $h(\hat{l}, \hat{f})$ is a value of primitive type
  - $h(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma(\hat{l}, \hat{f})$ (3.2.3.1.1) - hyp.1
- $h(\hat{l}, \hat{f})$ is not a value of primitive type
  - If $h(\hat{l}, \hat{f}) \neq l$ then $\Sigma'(h(\hat{l}, \hat{f})) = \Sigma(h(\hat{l}, \hat{f}))$, thus $h(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma(\hat{l}, \hat{f})$ (3.2.3.2.1) - hyp.3 + hyp.1
  - If $h(\hat{l}, \hat{f}) = l$ then $\lfloor \Sigma(h(\hat{l}, \hat{f})) \rfloor = \circ$, contradiction (3.2.3.2.2) - hyp.4 + hyp.6
  - $h(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma(\hat{l}, \hat{f})$ (3.2.3.2.3) - (3.2.3.2.1) + (3.2.3.2.2)
- $h(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma(\hat{l}, \hat{f})$ (3.2.3) - (3.2.3.1.1) + (3.2.3.1.3)
- $h'(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma'(\hat{l}, \hat{f})$ (3.2.4) - (3.2.1) + (3.2.2) + (3.2.3)

□

Lemma 18 (Store Delete)**.** *Let Σ and Σ' be heap typing environments, Γ and Γ' store typing environments, ρ a store, x a variable and f a field. Suppose that $\rho \vDash_\Sigma \Gamma$, $\Sigma' = \Sigma \backslash (\rho(x), f)$, $\Gamma' = \Gamma \backslash (x, f)$, $\lfloor \Sigma(\rho(x)) \rfloor = \circ$ and $NAOO_2(\rho, \Sigma)$. Then $\rho \vDash_{\Sigma'} \Gamma'$.*

Proof. Assume that $\rho \vDash_\Sigma \Gamma(hyp.1)$, $\Sigma' = \Sigma \backslash (\rho(x), f)(hyp.2)$, $\Gamma' = \Gamma \backslash (x, f)(hyp.3)$, $\lfloor \Sigma(\rho(x)) \rfloor = \circ(hyp.4)$ and $NAOO_2(\rho, \Sigma)(hyp.5)$

We want to prove that $\forall_{y \in dom(\Gamma')}\ \rho(y) \vDash_{\Sigma'} \Gamma'(y)$. Let us consider two cases:

Case $y = x$

- $\Gamma'(y) = \Gamma'(x), \rho(y) = \rho(x)$ (1)
- $\Sigma(\rho(x)) = \Gamma(x)$ (2) - hyp.1
- $\Sigma'(\rho(x)) = \Gamma'(x)$ (3) - (2) + hyp.2 + hyp.3
- $\rho(x) \vDash_{\Sigma'} \Gamma'(x)$ (4) - (3)
- $\rho(y) \vDash_{\Sigma'} \Gamma'(y)$ (5) - (1) + (4)

Case $y \neq x$

- $\Gamma'(y) = \Gamma(y)$ (1) - hyp.3

  Considering two subcases:
- $\rho(y)$ is a value of primitive type
  - $\rho(y) \vDash_\Sigma \Gamma(y)$ (2.1.1) - hyp.1
  - $\rho(y) \vDash_{\Sigma'} \Gamma'(y)$ (2.1.2) - (1) + (2.1.1)

- $\rho(y)$ is not a value of primitive type:
  - $\rho(y) \neq \rho(x)$ (2.2.1) - hyp.4 + hyp.5
  - $\Sigma'(\rho(y)) = \Sigma(\rho(y))$ (2.2.2) - (2.2.1) + hyp.4
  - $\rho(y) \vDash_{\Sigma} \Gamma(y)$ (2.2.3) - hyp.1
  - $\Sigma(\rho(y)) = \Gamma(y)$ (2.2.4) - (2.2.3)
  - $\Sigma'(\rho(y)) = \Gamma(y)$ (2.2.5) - (2.2.2) + (2.2.4)
  - $\rho(y) \vDash_{\Sigma'} \Gamma(y)$ (2.2.6) - (2.2.3) + (2.2.5)
  - $\rho(y) \vDash_{\Sigma'} \Gamma'(y)$ (2.1.2) - (1) + (2.2.6)

□

A.3 Object Creation

Lemma 19 (Heap New). *Let h and h' be heaps, Σ and Σ' heap typing environments and l a location. Suppose that $h \vDash \Sigma$, $h' = h[l \mapsto \{\}]$, $\Sigma' = \Sigma[l \mapsto \{\}^{\circ}]$ and $l \notin dom(h)$. Then $h' \vDash \Sigma'$.*

Proof. Assume that $h \vDash \Sigma$ (hyp.1), $h' = h[l \mapsto \{\}]$ (hyp.2), $\Sigma' = \Sigma[l \mapsto \{\}^{\circ}]$ (hyp.3) and $l \notin dom(h)$ (hyp.4).

Notice that $h' \vDash \Sigma'$ is equivalent to proving all the following conditions:

1. $dom(h') = dom(\Sigma')$
2. $\forall_{l \in dom(h')} dom(h'(l)) = dom(\Sigma'(l))$
3. $\forall_{l \in dom(h')} \forall_{f \in dom(h'(l))} h'(l, f) \vDash_{\Sigma'} \Sigma'(l, f)$

(1) It comes by:

- $dom(h') = dom(h) \cup \{l\}$ and $dom(\Sigma') = dom(\Sigma) \cup \{l\}$ (1.1.1) - hyp.2 + hyp.3
- $dom(h) = dom(\Sigma)$ (1.1.2) - hyp.1
- $dom(h') = dom(\Sigma')$ (1.1.3) - (1.1.1) + (1.1.2)

(2) For the second point we have two cases to consider:

Case $\hat{l} = l$:

- $dom(h'(\hat{l})) = \{\}$ and $dom(\Sigma'(\hat{l})) = \{\}$
  (2.1.1) - hyp.2 + hyp.3
- $dom(h'(\hat{l})) = dom(\Sigma'(\hat{l}))$ (2.1.2) - (2.1.1)

Case $\hat{l} \neq l$

- $dom(h'(\hat{l})) = dom(h(\hat{l}))$ and $dom(\Sigma'(\hat{l})) = dom(\Sigma(\hat{l}))$ (2.1) - hyp.2 + hyp.3
- $dom(h(\hat{l})) = dom(\Sigma(\hat{l}))$ (2.2) - hyp.1
- $dom(h'(\hat{l})) = dom(\Sigma'(\hat{l}))$ (2.3) - (2.1) + (2.2)

(3) We have only one case to consider since $dom(h'(l)) = \{\}$:

Case $\hat{l} = l$ It does not apply since $dom(h(\hat{l})) = dom(\Sigma(\hat{l})) = \{\}$

Case $\hat{l} \neq l$

- $h'(\hat{l}, \hat{f}) = h(\hat{l}, \hat{f})$ (3.2.1) - hyp.2
- $\Sigma'(\hat{l}, \hat{f}) = \Sigma(\hat{l}, \hat{f})$ (3.2.2) - hyp.3

  Thus, considering two subcases:

- $h(\hat{l}, \hat{f})$ is a value of primitive type
  - $h(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma(\hat{l}, \hat{f})$ (3.2.3.1.1) - hyp.1
- $h(\hat{l}, \hat{f})$ is not a value of primitive type
  - If $h(\hat{l}, \hat{f}) \neq l$ then $\Sigma'(h(\hat{l}, \hat{f})) = \Sigma(h(\hat{l}, \hat{f}))$, thus $h(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma(\hat{l}, \hat{f})$

    (3.2.3.2.1) - hyp.3 + hyp.1
  - If $h(\hat{l}, \hat{f}) = l$, contradiction

    (3.2.3.2.2) - hyp.4
  - $h(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma(\hat{l}, \hat{f})$ (3.2.3.2.3) - (3.2.3.2.1) + (3.2.3.2.2)
- $h(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma(\hat{l}, \hat{f})$ (3.2.3) - (3.2.3.1.1) + (3.2.3.1.3)
- $h'(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma'(\hat{l}, \hat{f})$ (3.2.4) - (3.2.1) + (3.2.2) + (3.2.3)

□

Lemma 20 (Store New)**.** *Let* Σ *and* Σ' *be heap typing environments,* Γ *and* Γ' *store typing environments,* ρ *and* ρ' *stores and* x *a variable. Suppose that* $\rho \vDash_\Sigma \Gamma$*,* $\rho' = \rho[x \mapsto l]$*,* $\Sigma' = \Sigma[\rho'(x) \mapsto \{\}^\circ]$*,* $\Gamma' = \Gamma[x \mapsto \{\}^\circ]$ *and* $l \notin dom(\Sigma)$*. Then* $\rho' \vDash_{\Sigma'} \Gamma'$*.*

Proof. Assume that $\rho \vDash_\Sigma \Gamma(hyp.1)$, $\rho' = \rho[x \mapsto l](hyp.2)$ $\Sigma' = \Sigma[\rho'(x) \mapsto \{\}^\circ](hyp.3)$, $\Gamma' = \Gamma[x \mapsto \{\}^\circ]$ and $l \notin dom(\Sigma)(hyp.5)$.

We want to prove that $\forall_{y \in dom(\Gamma')}\ \rho(y) \vDash_{\Sigma'} \Gamma'(y)$. Let us consider two cases:

Case $y = x$

- $\Gamma'(y) = \Gamma'(x), \rho'(y) = \rho'(x)$ (1)
- $\Sigma'(\rho'(x)) = \Gamma'(x)$ (2) - hyp.2 + hyp.3 + hyp.4
- $\rho'(x) \vDash_{\Sigma'} \Gamma'(x)$ (3) - (2)
- $\rho'(y) \vDash_{\Sigma'} \Gamma'(y)$ (4) - (1) + (3)

Case $y \neq x$

- $\Gamma'(y) = \Gamma(y)$, $\rho'(y) = \rho(y)$ (1) - hyp.2 + hyp.4

  Considering two subcases:
- $\rho(y)$ is a value of primitive type

- $\rho(y) \models_{\Sigma} \Gamma(y)$ (2.1.1) - hyp.1
- $\rho'(y) \models_{\Sigma'} \Gamma'(y)$ (2.1.2) - (1) + (2.1.1)

- $\rho(y)$ is not a value of primitive type:
  - $\rho(y) \neq \rho'(x)$ (2.2.1) - hyp.5
  - $\Sigma'(\rho(y)) = \Sigma(\rho(y))$ (2.2.2) - (2.2.1) + hyp.4
  - $\rho(y) \models_{\Sigma} \Gamma(y)$ (2.2.3) - hyp.1
  - $\Sigma(\rho(y)) = \Gamma(y)$ (2.2.4) - (2.2.3)
  - $\Sigma'(\rho(y)) = \Gamma(y)$ (2.2.5) - (2.2.2) + (2.2.4)
  - $\rho(y) \models_{\Sigma'} \Gamma(y)$ (2.2.6) - (2.2.3) + (2.2.5)
  - $\rho'(y) \models_{\Sigma'} \Gamma'(y)$ (2.1.2) - (1) + (2.2.6)

□

A.4 Object Closing

Lemma 21 (Heap Close). *Let h ba an heap, Σ and Σ' heap typing environments, ρ a store and l a location. Suppose that $h \models \Sigma$, $\Sigma' = \Sigma[l \mapsto \Sigma(l)^{\bullet}]$, $NAOO_1(h, \Sigma)$ and $\lfloor \Sigma(l) \rfloor = \circ$. Then $h \models \Sigma'$, .*

Proof. Assume that $h \models \Sigma$ (hyp.1), $\Sigma' = \Sigma[l \mapsto \Sigma(l)^{\bullet}]$ (hyp.3), $NAOO_1(h, \Sigma)$ (hyp.4) and $\lfloor \Sigma(l) \rfloor = \circ$ (hyp.6).

Notice that $h' \models \Sigma'$ is equivalent to provent all the following conditions:

1. $dom(h') = dom(\Sigma')$
2. $\forall_{l \in dom(h')} dom(h'(l)) = dom(\Sigma'(l))$
3. $\forall_{l \in dom(h')} \forall_{f \in dom(h'(l))} h'(l, f) \models_{\Sigma'} \Sigma'(l, f)$

(1) It comes by:

- $dom(\Sigma') = dom(\Sigma)$ (2.1.1) - hyp.3
- $dom(h) = dom(\Sigma)$ (2.1.2) - hyp.1
- $dom(h) = dom(\Sigma')$ (2.1.3) - (2.1.1) + (2.1.2)

(2) Consider any $\hat{l} \in dom(h)$, then:

- $dom(\Sigma'(\hat{l})) = dom(\Sigma(\hat{l}))$ (2.1) - hyp.3
- $dom(h(\hat{l})) = dom(\Sigma(\hat{l}))$ (2.2) - hyp.1
- $dom(h(\hat{l})) = dom(\Sigma'(\hat{l}))$ (2.3) - (2.1) + (2.2)

(3) Let $(\hat{l}, \hat{f}) \in dom(h)$. We have two cases to consider:
Case $h(\hat{l}, \hat{f}) \neq l$:

- $h(\hat{l}, \hat{f}) \models_{\Sigma} \Sigma(\hat{l}, \hat{f})$ (3.1.1) - hyp.1

- $\Sigma'(\hat{l}, \hat{f}) = \Sigma(\hat{l}, \hat{f})$ (3.1.2) - hyp.2
- $h(\hat{l}, \hat{f}) \vDash_{\Sigma'} \Sigma'(\hat{l}, \hat{f})$ (3.1.3) - (3.1.1) + (3.1.2)

Case $(\hat{l}, \hat{f}) = l$

- $\lfloor \Sigma'(\hat{l}, \hat{f}) \rfloor = \bullet$, contradiction (3.2.1) - hyp.4 + hyp.6

□

Lemma 22 (Store Close). *Let Σ and Σ' be heap typing environments, Γ and Γ' store typing environments, ρ a store and x a variable. Suppose that $\rho \vDash_\Sigma \Gamma$, $\Sigma' = \Sigma[\rho(x) \mapsto \Sigma(\rho(x))^\bullet]$, $\Gamma' = \Gamma[x \mapsto \Gamma(x)^\bullet]$, $\lfloor \Sigma(\rho(x)) \rfloor = \circ$ and $NAOO_2(\rho, \Sigma)$. Then $\rho \vDash_{\Sigma'} \Gamma'$.*

Proof. Assume that $\rho \vDash_\Sigma \Gamma(hyp.1)$, $\Sigma' = \Sigma[\rho(x) \mapsto \Sigma(\rho(x))^\bullet](hyp.2)$, $\Gamma' = \Gamma[x \mapsto \Gamma(x)^\bullet](hyp.3)$, $\lfloor \Sigma(\rho(x)) \rfloor = \circ(hyp.4)$ and $NAOO_2(\rho, \Sigma)(hyp.5)$

We want to prove that $\forall_{y \in dom(\Gamma')}\ \rho(y) \vDash_{\Sigma'} \Gamma'(y)$. Let us consider two cases:

Case $y = x$

- $\Gamma'(y) = \Gamma'(x), \rho(y) = \rho(x)$ (1)
- $\Sigma(\rho(x)) = \Gamma(x)$ (2) - hyp.1
- $\Sigma'(\rho(x)) = \Gamma'(x)$ (3) - (2) + hyp.2 + hyp.3
- $\rho(x) \vDash_{\Sigma'} \Gamma'(x)$ (4) - (3)
- $\rho(y) \vDash_{\Sigma'} \Gamma'(y)$ (5) - (1) + (4)

Case $y \neq x$

- $\Gamma'(y) = \Gamma(y)$ (1) - hyp.3

  Considering two subcases:

- $\rho(y)$ is a value of primitive type
  - $\rho(y) \vDash_\Sigma \Gamma(y)$ (2.1.1) - hyp.1
  - $\rho(y) \vDash_{\Sigma'} \Gamma'(y)$ (2.1.2) - (1) + (2.1.1)
- $\rho(y)$ is not a value of primitive type:
  - $\rho(y) \neq \rho(x)$ (2.2.1) - hyp.4 + hyp.5
  - $\Sigma'(\rho(y)) = \Sigma(\rho(y))$ (2.2.2) - (2.2.1) + hyp.4
  - $\rho(y) \vDash_\Sigma \Gamma(y)$ (2.2.3) - hyp.1
  - $\Sigma(\rho(y)) = \Gamma(y)$ (2.2.4) - (2.2.3)
  - $\Sigma'(\rho(y)) = \Gamma(y)$ (2.2.5) - (2.2.2) + (2.2.4)
  - $\rho(y) \vDash_{\Sigma'} \Gamma(y)$ (2.2.6) - (2.2.3) + (2.2.5)
  - $\rho(y) \vDash_{\Sigma'} \Gamma'(y)$ (2.1.2) - (1) + (2.2.6)

□

Appendix B

Other Lemmas

Lemma 23 (Weakening Lemma)**.** *Let h be an heap, ρ a store, Σ a heap typing environment and Γ and Γ' store typing environments. Suppose that $h, \rho \vDash \Sigma, \Gamma$ then, for any Γ' we have that $h, \rho \vDash \Sigma, \Gamma \sqcap \Gamma'$.*

Proof. Assume that $h, \rho \vDash \Sigma, \Gamma$ (hyp.1) and let Γ' be any typping environment.
We start by noticing that $h, \rho \vDash \Sigma, \Gamma \sqcap \Gamma' \Leftrightarrow \rho \vDash_\Sigma \Gamma \sqcap \Gamma'$ and $h \vDash \Sigma$.
Then we have to prove that $\rho \vDash_\Sigma \Gamma \sqcap \Gamma'$ and $h \vDash \Sigma$.
From hyp.1 we conclude that $h \vDash \Sigma$. Now it remains to prove that $\rho \vDash_\Sigma \Gamma \sqcap \Gamma'$, which requires proving that $\rho(x) \vDash_\Sigma \Gamma \sqcap \Gamma'(x)$ for all $x \in dom(\Gamma \sqcap \Gamma')$. Observing that $\Gamma \sqcap \Gamma' \subseteq \Gamma$, it follows that $\rho(x) \vDash_\Sigma \Gamma \sqcap \Gamma'(x)$. □

Lemma 24 (Satisfaction Uniqueness)**.** *Let Σ be an heap typing environment, v be a value, and τ_1 and τ_2 types. Suppose that $v \vDash_\Sigma \tau_1$ and $v \vDash_\Sigma \tau_2$. Then $\tau_1 = \tau_2$.*

Proof. Assume that $v \vDash_\Sigma \tau_1 (hyp.1)$ and $v \vDash_\Sigma \tau_2 (hyp.2)$. We have four cases to consider:
Case v is a $number$ (hyp.3)

- $\tau_1 = number$ (1.1) - hyp.1 + hyp.3
- $\tau_2 = number$ (1.2) - hyp.2 + hyp.3
- $\tau_1 = \tau_2$ (1.3) - (1.1) + (1.2)

Case v is a $string$ (hyp.3)

- $\tau_1 = string$ (2.1) - hyp.1 + hyp.3
- $\tau_2 = string$ (2.2) - hyp.2 + hyp.3
- $\tau_1 = \tau_2$ (3.3) - (2.1) + (2.2)

Case v is a $boolean$ (hyp.3)

- $\tau_1 = boolean$ (3.1) - hyp.1 + hyp.3
- $\tau_2 = boolean$ (3.2) - hyp.2 + hyp.3
- $\tau_1 = \tau_2$ (3.3) - (3.1) + (3.2)

Case v is a location l (hyp.3)

- $\tau_1 = \Sigma(l)$ (4.1) - hyp.1 + hyp.3
- $\tau_2 = \Sigma(l)$ (4.2) - hyp.2 + hyp.3

- $\tau_1 = \tau_2$ (4.3) - (4.1) + (4.2)

□

Lemma 25 (Closed values)**.** *Let τ be a type, v a value and Σ a heap typing environment. Suppose that* $\mathsf{Closed}(\tau)$ *and* $v \vDash_\Sigma \tau$*. Then* $\mathsf{Closed}_\Sigma(v)$*.*

Proof. Assume that $\mathsf{Closed}(\tau)(hyp.1)$ and $v \vDash_\Sigma \tau(hyp.2)$. Therefore we have two cases to consider:
Case τ is a primitive type (hyp.3)

- v is of primitive type (1.1) - hyp.2 + hyp.3
- $Type(v) = \tau$
- $\mathsf{Closed}_\Sigma(v)$ (1.2) - (1.1)

Case τ is not a primitive type (hyp.3)

- v is location
- $\Sigma(v) = \tau$ (1.1) - (1hyp.2 + hyp.3
- $\mathsf{Closed}_\Sigma(v)$ (1.2) - (1.1)

□