UNIVERSITY OF NICE - SOPHIA ANTIPOLIS
DOCTORAL SCHOOL STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION
ET DE LA COMMUNICATION

P H D T H E S I S

to obtain the title of

Doctor of Computer Science

of the University of Nice - Sophia Antipolis

To Be Defended by

José Fragoso Santos

Enforcing Secure Information Flow in Client-Side Web Applications

Vers l'Établissement du Flux d'Information Sûr dans les Applications Web Côté Client

Advised by Tamara Rezk and Ana Almeida Matos

prepared at INRIA Sophia Antipolis, Team Indes

to be defended on December 8th, 2014

Jury :

President :	Cédric Fournet	-	Microsoft Research
Reviewers :	Peter Thiemann	-	University of Freiburg
	David A. Naumann	-	Stevens Institute of Technology
Examiners :	Vasco T. Vasconcelos	-	University of Lisbon
Advisors :	Tamara Rezk	-	Inria
	Ana Almeida Matos	-	University of Lisbon
Invited :	Gérard Boudol	-	Inria

Abstract

During the last decade, Web applications have evolved from static pages presented by Web servers which centralised all computations to multi-tier applications in which computations are shared between the client and the server. In addition to this, current client-side Web applications often combine code dynamically loaded from different origins to create new functionalities. As it happens, this architectural style allows for an ever widening spectrum of security vulnerabilities, since malicious third-party scripts may compromise the security of the whole system. In this scenario, many attacks arise at the application level, and can thus be tackled by means of programming language design and analysis techniques, such as static analysis or program instrumentation.

In this thesis, we address the issue of enforcing confidentiality and integrity policies in the context of client-side Web applications. Since most Web applications are developed in the JavaScript programming language, we study static, dynamic, and hybrid enforcement mechanisms for securing information flow in Core JavaScript — a fragment of JavaScript that retains its defining features. Specifically, we propose:

1. a monitored semantics for dynamically enforcing secure information flow in Core JavaScript as well as a source-to-source transformation that inlines the proposed monitor,

2. a type system that statically checks whether or not a program abides by a given information flow policy, and

3. a hybrid type system that combines static and dynamic analyses in order to accept more secure programs than its fully static counterpart.

Most JavaScript programs are designed to be executed in a browser in the context of a Web page. These programs often interact with the Web page in which they are included via a large number of external APIs provided by the browser. The execution of these APIs usually takes place outside the perimeter of the language. Hence, any realistic analysis of client-side JavaScript must take into account possible interactions with external APIs. To this end, we present a general methodology for extending security monitors to take into account the possible invocation of arbitrary APIs and we apply this methodology to a representative fragment of the DOM Core Level 1 API that captures DOM-specific information flows.

Résumé

Au cours de la dernière décennie les applications Web sont passées d'une architecture dans laquelle le serveur Web était chargé de tous les calculs à une architecture multi-levels dont les calculs sont partagés entre le client et le serveur. De plus, actuellement les applications Web côté client sont souvent le résultat d'une combinaison de plusieurs scripts issus d'origines différentes. Ce style architectural expose les applications Web à un très large éventail de failles de sécurité puisque des scripts tiers malveillants peuvent mettre en cause la sécurité de tout le système. Dans ce scénario, plusieurs attaques surgissent au niveau de l'application. Par conséquent, ces attaques peuvent être surmontés à travers des techniques de conception ainsi que de l'analyse des langages de programmation, comme l'analyse statique et l'instrumentation du code.

Nous nous intéressons à la mise en œuvre des politiques de confidentialité et d'intégrité des données dans le contexte des applications Web côté client. Étant donné que la plupart des applications Web est développée en JavaScript, on propose des mécanismes statiques, dynamiques et hybrides pour sécuriser le flux d'information en Core JavaScript - un fragment de JavaScript qui retient ses caractéristiques fondamentales. Nous étudions en particulier:

1. une sémantique à dispositif de contrôle afin de garantir dynamiquement le respect des politiques de sécurité en Core JavaScript aussi bien qu'un compilateur qui instrumente un programme avec le dispositif de contrôle proposé,

2. un système de types qui vérifie statiquement si un programme respecte une politique de sécurité donnée,

3. un système de types hybride qui combine des techniques d'analyse statique à des techniques d'analyse dynamique afin d'accepter des programmes surs que sa version purement statique est obligée de rejeter.

La plupart des programmes JavaScript s'exécute dans un navigateur Web dans le context d'une page Web. Ces programmes interagissent avec la page dans laquelle ils sont inclus parmi des APIs externes fournies par le navigateur. Souvent, l'execution d'une API externe dépasse le périmètre de l'interprète du langage. Ainsi, une analyse réaliste des programmes JavaScript côté client doit considérer l'invocation potentielle des APIs externes. Pour cela, on présente une méthodologie générale qui permet d'étendre des dispositifs de contrôle de sécurité afin qu'ils prennent en compte l'invocation potentielle des APIs externes et on applique cette méthodologie à un fragment important de l'API DOM Core Level 1.

Contents

List of Figures

Chapter 1

Introduction

Contents

Web applications hold a prominent spot in the Internet of today. They are increasingly used by people in their everyday lives to accomplish all sorts of tasks, including e-mailing, word processing, online banking and shopping, and many, many other. While some of these applications do not necessarily mandate a high level of security, there are those, such as online banking, for which it is of paramount importance. Security of Web applications is, therefore, an important and highly applicable research topic. And, in order to be able to address it properly, we begin by taking a closer look at their general structure.

Most Web applications are composed of several different programs, called scripts, which do not necessarily share the same origin. Some of these scripts can even be loaded from third-party code providers at runtime; this is the case, for example, when it comes to online advertisements. The code whose origin coincides with that of the Web page is called the *integrator*, whereas each external script is called a *gadget*. Using gadgets in a Web application is not mandatory, but if any are involved, it is then the job of the integrator to patch them all together in order to generate the Web application. The resulting Web application is called a *Web mashup*. The programming language that is typically used for the implementation of Web mashups is JavaScript [5th edition of ECMA 262 2011] — a widely used programming language supported by all of the major browsers.

What can be said about the security issues that are raised by the use of external gadgets in a Web application? The fact most pertinent to this question is that gadgets can be loaded at runtime and can even depend on the input given by the user. This is commonplace, for instance, for online advertisements, which are loaded from ad servers that use various data mining techniques in order to determine which advertisements should be displayed to which user. Therefore, it is impossible for the developer of such Web applications to know *a priori* which third-party code will be executed. This architectural style of modern Web applications can raise serious security issues — malicious third-party programs can compromise the integrity and confidentiality of the user's resources. Illustratively, a recent study by Jang et al. [Jang 2010] has shown that many Websites, including some in the Alexa global top-100, exhibit privacy-violating security vulnerabilities.

In light of the current security-critical situation, a common interest exists between Web application developers and users alike in the enforcement of isolation properties that guarantee that confidential resources are not leaked to untrusted parties and that high-integrity resources are not modified based on low-integrity data coming from untrusted gadgets. In fact, the central concept in the Web application security model, the Same Origin Policy (SOP) [Barth 2011], was designed to provide precisely this type of guarantees. Roughly, this policy states that a script

loaded from one origin is not allowed to access or modify resources obtained from another origin. Here, as in [Yang 2013], we refer to this definition as the *strict* SOP. While a full implementation of the strict SOP would definitely solve most of the security issues that wreak havoc on modern Web applications, it would, unfortunately, also severely constrain one of their essential features, that being the interaction between scripts of different origins within a Web page. As it is, in order to allow for cross-origin communication, the browser security model includes many exceptions to the strict SOP. For instance, current browsers allow for the inclusion of an external gadget in a Web page in two different ways:

- either through the creation of a *script* node that is not subject to the Same Origin Policy, meaning that the included gadget is executed in the same environment as the integrator and has read/write access to all of its resources;

- or through the creation of an *iframe* node, subject to the Same Origin Policy, with the included gadget executed in a separate environment, commonly referred to as a *sandbox*, from which it does not have direct access to the integrator's resources[1].

Since the Same Origin Policy is, in fact, implemented in current browsers, it is possible to take advantage of it when designing secure Web applications. This can be accomplished by the developer [Barth 2009], or automatically [Louw 2012, Luo 2012]. However, the complexity of the API for interframe communication often makes it hard and cumbersome to manually sandbox the execution of external gadgets.

Making use of the SOP does not guarantee, by itself, security of Web applications as there are security issues that lay beyond its scope. Even if we sandbox the execution of a gadget (preventing it from **actively** compromising the integrity and confidentiality of the user's resources), the integrator can inadvertently leak confidential information to that gadget or corrupt high-integrity resources using data originating from that gadget. In other words, a sandboxing mechanism can allow the integrator to use the API for interframe communication as an *escape hatch* for sending/receiving **arbitrary** information to/from external gadgets. Hence, this type of mechanism is only fit to enforce security policies such that the integrator is allowed to declassify/endorse everything it sends/receives to/from external gadgets, as in *delimited release* [Sabelfeld 2003b]. In order to provide stronger security guarantees, one needs to resort to more powerful techniques than simply sandboxing the execution of third-party code. In particular, one needs to control the information flows that take place within the code of the integrator in order to decide which information can be securely sent to which gadget and/or which resources can be modified by which gadget-based information.

Another problem of SOP-based sandboxing mechanisms for Web applications is that their precision is constrained by the precision of the SOP. In fact, it has been observed that *"the SOP is merely a highly restrictive Information Flow Control policy in which flows between origins are denied"* [Yang 2013]. By using the SOP as a means for securing Web applications, one is essentially constraining the level of granularity of the security policies that can be enforced. Concretely, when using the SOP in the design of a security mechanism, one is forced to view each origin as a security principal in the system [Magazinius 2010a]. While it is possible to assign different security credentials to different sets of principals/origins, it is not possible to assign different security credentials to the same principal/origin depending on how it uses the information that it is given. For instance, suppose that we would like to express that a given gadget can have access to certain confidential information as long as it does not send it to the server from which it was issued. The only way to enforce this type of policy is through the use of an Information Flow Control (IFC) mechanism.

[1] In this case, communication is still possible via the PostMessage API [Barth 2009].

We support the view that *"Information Flow Control is a good fit for whole-browser security"* [Yang 2013], as it can perfectly capture the SOP, but also express more fine-grained security policies whose enforcement eliminates security vulnerabilities in current Web applications, while at the same time allowing for the flexibility of cross-origin communication.

1.1 Securing Information Flow in a Core of JavaScript

Noninterference [Goguen 1982] is a class of properties that have been classically used to reason about how the execution of a program propagates or how it generates dependencies between the resources it manipulates. The problem of enforcing secure information flow is essentially a problem of preventing the execution of programs that can potentially create illegal dependencies between the resources they operate on. For instance, confidentiality-wise, a program is secure if its execution does not entail the creation of dependencies between public outputs and secret inputs. In other words, secret inputs cannot influence public outputs. Analogously, integrity-wise, a program is secure if its execution does not entail the creation of dependencies between high-integrity outputs and low-integrity inputs. In other words, low-integrity inputs cannot influence high-integrity outputs. Thus, noninterference provides the mathematical foundation for reasoning precisely about secure information flow and, in fact, it has been largely used [Hedin 2011, Sabelfeld 2003a] to formally express the absence of security leaks for a wide variety of programming languages ranging from functional (e.g. [Pottier 2002]) to object-oriented (e.g. [Banerjee 2002]) in both sequential (e.g. [Volpano 1996]) and concurrent settings (e.g. [Almeida Matos 2009]).

The stating of the dependencies that the execution of a program can legally generate generally betakes a certain degree of abstraction. It is not always possible or even desirable to talk about the actual resources that a program manipulates. Instead, it is often more convenient to reason about classes of resources that mandate the same degree of security. We can, therefore, see an *information flow policy* as a partially ordered set of security levels together with a mapping establishing the security levels of the resources on which the program operates. This mapping, which we call a *security labelling*, can be interpreted as an abstraction of the concrete resources of the program [Cousot 1977, Hunt 2006]. Having established a security policy, we say that, given two resources A and B, an information flow from A to B is legal if the security level of B is higher than or equal to the level of A. Whenever two levels L_A and L_B are in the order relation ($L_A \sqsubseteq L_B$), it means that the use of information at level L_B is at least as restrictive as the use of information at level L_A. More restrictive security levels correspond to higher confidentiality and lower integrity, since high-confidentiality resources are not allowed to affect low-confidentiality resources and low-integrity resources are not allowed to affect high-integrity resources. Intuitively, information is allowed to move up in the partially ordered set of security levels but not down. For convenience, we assume that the partially ordered set of security levels constitutes a lattice [Davey 2002], meaning that the *least upper bound* (*lub*) and the *greatest lower bound* (*glb*) between any two security levels are always defined.

In the context of information flow research, the enforcement of integrity policies [Biba 1977, Li 2003] can be viewed as the dual problem of the enforcement of confidentiality policies. Hence, in the remainder of the thesis we shall always refer to confidentiality policies, while the application of the proposed mechanisms to the enforcement of integrity policies would be straightforward.

Confidentiality-wise, given a concrete program state, a security labelling defines what part of that state is visible at each security level. Hence, if a security labelling is too coarse, it will declare invisible resources that should be visible. In this sense, coarse security policies inevitably cause secure programs not to abide by noninterference and therefore be rejected by

sound enforcement mechanisms. Thus, it is vital that the *"abstractions made in the attacker model be adequate with respect to potential attacks"* [Sabelfeld 2003a]. In other words, security policies should be rich enough to capture the various types of attacks coming from the language, thus adequately reflecting its expressive power. The question to be answered is: *"What can an attacker see using the constructs of the language?"* The answer to this question is not always trivial, since not only are the contents of a program state visible to an attacker, but also the structure of these contents. For instance, in JavaScript, as in other object oriented languages, a program can inspect the values associated with the fields of an object. However, unlike most other languages, JavaScript also allows a program to check which are the fields that an object defines.

In this thesis, we begin by defining noninterference for Core JavaScript - a fragment of JavaScript that retains its defining features. Particularly, the proposed definition of noninterference makes use of security policies that reflect the specificities of the language (such as the fact that programs can check the existence of object fields). We then study different types of mechanisms (both static and dynamic) to enforce variations of the proposed security property.

The dynamic nature of JavaScript renders it an exceedingly difficult language to statically analyse [Maffeis 2009]. Consequently, sound static analyses for JavaScript are in general largely over-conservative and reject many secure programs. Contrastingly, dynamic analyses are normally less conservative than static analyses, but impose a performance overhead that is often non-negligible [Hedin 2014]. In this thesis, we propose: **(1)** a purely dynamic monitor that enforces secure information flow in Core JavaScript as well as source-to-source transformation that inlines the monitor, **(2)** a type system that statically checks whether or not a Core JavaScript program abides by a given information flow policy, and finally **(3)** a hybrid type system that combines static and dynamic analyses in order to accept more programs than its fully static counterpart. This hybrid type system leverages the combination of static and runtime analysis to overcome some of the disadvantages of purely static and purely dynamic approaches.

1.2 Securing Information Flow in the Browser

Although JavaScript can be used as general-purpose programming language, most JavaScript programs are conceived to be executed in a browser in the context of a Web page. These programs often interact with the Web page in which they are included *via* the *Application Programming Interfaces* (APIs) provided by the browser, such as the Document Object Model API (DOM API), the XMLHttpRequest API, or the W3C Geolocation API. The semantics of these APIs often escapes the semantics of JavaScript in the sense that, since they are not implemented in JavaScript, their execution is not managed by the JavaScript engine, but rather by a dedicated and separate module of the browser [Grosskurth 2005]. Thus, a realistic analysis of client-side JavaScript code must include an analysis of the APIs that the targeted programs are supposed to use. However, the continuous emergence and heterogeneity of different APIs [Guha 2012] renders the problem of precise reasoning about JavaScript client-side code extremely challenging. This is particularly relevant in the context of information flow security. Hence, to tackle this problem, this thesis presents a general methodology for extending security monitors in order for them to take into account the possible invocation of arbitrary external APIs. We then apply this methodology to extend our information flow monitor for Core JavaScript as well as the corresponding source-to-source program transformation.

The DOM API [W3C Recommendation 2000, W3C Recommendation 2005] occupies a central role among the APIs that browsers make available for JavaScript programs. Indeed, every modern browser includes a DOM implementation that manages the integration between JavaScript and the user interface of the browser. More concretely, JavaScript programs use the

DOM API to interact with the HTML page that the browser displays on the screen — to change or simply access the content of the page as well as the input coming from the user. In a certain sense, one can also view the DOM as the data structure corresponding to the "in memory" counterpart of the displayed HTML page. In fact, the displayed document is represented in the DOM API as a tree structure, whose nodes correspond to the various types of content in the document.

Unsurprisingly, malicious programs can use the DOM to encode illegal information flows [Russo 2009]. Hence, to make sure that a JavaScript program is secure, one must analyse how it interacts with the Web page in which it is included via the DOM API. In this thesis, we present a group of monitor extensions for handling an important fragment of the DOM Core Level 1 API, that we call Core DOM. There, as in the DOM API, DOM nodes are treated as first-class values. Using this, we are able to construct an information flow control mechanism that is more fine-grained than the previous approaches in the literature [Russo 2009]. We also introduce methods and properties for modelling the behaviour of *live collections* — a special type of data structure in the DOM Core Level 1 API. We show that live collections effectively augment the observational power of an attacker and we show how to monitor their use in order to enforce secure information flow.

1.3 Contributions and Outline

In a nutshell, the original contributions of this thesis are the following:

- A new information flow monitor-inlining transformation for a core of JavaScript that retains its defining features, such as prototype-based inheritance, extensible objects, constructs for checking the existence of object fields, and unusual interactions between the binding of variables and the binding of properties;
- A hybrid type system for checking whether or not a Core JavaScript program abides by a given information flow policy that combines static and dynamic analysis to avoid rejecting programs that are in fact secure;
- A general methodology for extending information flow monitors to take into account the execution of arbitrary APIs, possibly outside of the perimeter of the modelled language;
- An information flow monitor that handles an important fragment of the DOM Core Level 1 API, including live collections, which had not been formally studied so far in the context of Information Flow Control (IFC) research.

The outline of the thesis is as follows:

- Chapter 2 presents the fragment of JavaScript that is studied in this thesis, which we call Core JavaScript. This core takes into account the defining features of the language mentioned above.
- Chapter 3 defines what it means for a Core JavaScript program to be noninterferent. The proposed definition of noninterference makes use of security policies that accurately capture the expressiveness of the language by taking into account its main specificities.
- Chapter 4 first presents a monitor that dynamically enforces secure information flow for Core JavaScript as well as a source-to-source transformation that inlines the monitor. The presented monitor is proven sound, that is noninterferent, and the compiler is proven correct with respect to the monitor. Therefore, we ensure that, after compilation, only

secure executions are allowed to go through, as potentially illegal executions are caused to diverge by the inlined runtime enforcement mechanism.

- Chapter 5 first presents a purely static type system for securing information flow in Core JavaScript. Using this type system as a starting point, we develop a hybrid type system for information flow control in Core JavaScript. Unlike purely static type systems, which only accept programs when they can guarantee that all possible execution paths are secure, the hybrid type system we propose infers a set of assertions under which a program can be securely accepted and instruments it so as to dynamically check whether these assertions hold. By deferring rejection to runtime, this hybrid version is able to typecheck secure programs that purely static type systems cannot accept.

- Chapter 6 proposes a methodology for extending sound JavaScript information flow monitors. This methodology allows us to enforce compliance of a monitor with the proposed noninterference property in a modular way. Thus, proving that a monitor is noninterferent after extending it with a new API only requires the proof that the API itself is noninterferent. We apply this methodology to extend our information flow monitor for Core JavaScript. Furthermore, this chapter presents an extension of the information flow monitor-inlining compiler defined in Chapter 4 that additionally takes into account the invocation of arbitrary APIs.

- Chapter 7 presents a group of monitor extensions for handling a fragment of the DOM Core Level 1 API, that we call Core DOM API. In the Core DOM API, as in the DOM API, tree nodes are treated as first-class values. We take advantage of this feature in order to design an information flow control mechanism that is more fine-grained than the previous approaches in the literature [Russo 2009]. Furthermore, we extend Core DOM with additional API methods that model the behaviour of *live collections*, a type of data structure present in the DOM Core Level 1 API that exhibits a very unusual semantics. We show that the use of live collections effectively augments the observational power of an attacker and we provide monitor extensions to tackle these newly introduced forms of information leaks.

1.4 Publications

While certain elements of this thesis remain unpublished to this day, the remaining parts have previously appeared in the following publications:

- Fragoso Santos, José and Rezk, Tamara. An Information Flow Monitor Inlining Compiler For Securing a Core of JavaScript. IFIP SEC, 2014
  This paper presents a version of the information flow monitor-inlining compiler here introduced in Chapter 4, which was, to the best of our knowledge, the first of this type of compilers designed for a JavaScript-like language. The information flow monitor used in the paper as well as its respective source-to-source transformation differ from those of the thesis in that they consider a smaller subset of JavaScript. Namely, they do not include neither the `in` nor the `delete` program constructs, which we do include here. Since the these constructs effectively augment the observational power of an attacker, their inclusion in the targeted fragment of the language required changing the way program resources are labeled.

- Almeida-Matos, Ana, Fragoso Santos, José and Rezk, Tamara. An Information Flow Monitor for a Core of DOM – Introducing references and live primitives. TGC, 2014

The paper presents a novel, purely dynamic, flow-sensitive monitor for securing information flow in an imperative language extended with DOM-like tree operations, which is proven sound with respect to a standard notion of noninterference for monitors. The monitor extensions presented in Chapter 6 partially coincide with the language primitives for operating on tree nodes studied in this paper. The main difference is that here we study these operation in the context of Core JavaScript, while in the paper they were studied in the context of a simple WHILE language.

Chapter 2

Core JavaScript

Contents

In a nutshell, JavaScript is an object-based, untyped, language which supports closures and prototype-based inheritance [3rd edition of ECMA 262 1999, 5th edition of ECMA 262 2011]. Indeed, objects are the central datatype of JavaScript. But, in contrast to class-based languages where the fields of an object are restricted by the class to which it belongs (which is statically specified), a JavaScript object is an unrestricted partial mapping from strings to values. The strings in the domain of an object are called its *properties*. In JavaScript there are two types of objects: those that are defined by the programmer and those that are provided by the language runtime. The latter are called *internal objects*.

JavaScript is an object-based language. However, there are no classes. Instead, every *non-native* object has a prototype from which it can *inherit* properties. Prototypes are also objects. Hence, *prototypical inheritance* is a form of delegation, in the sense that an object dispatches to its prototype the requests that it does not know how to handle. For instance, in order to look-up the value of a property `"xpto"` of an object bound to a variable `o`, the JavaScript engine first checks whether `"xpto"` belongs to the set of properties of the object bound to `o`. If so, the property look-up yields the value with which that object associates property `"xpto"`. Otherwise, the engine checks whether the prototype of that object defines a property named `"xpto"`, and so forth. The sequence of objects that can be accessed from a given object through the inspection of the respective prototypes is called a *prototype-chain*.

JavaScript features first-class functions. Functions can be used in three different ways: as usual functions, as *methods*, or as *constructors*. When assigning a function to a property of an object, the function becomes a *method* of the object. Every method accessible to an object through its prototype-chain can be called as a method of that object. Concretely, when calling a function as a method, the keyword this is bound to the *receiver object*, that is, the object on which the method was called. For instance, suppose that the object bound to `o` has access to a property

$x, y_1, \cdots, y_n \in$ `Var`	::=	`foo` | `bar` | `baz` | $\cdots$	% Identifiers
$m, p \in$ `Str`	::=	`"foo"` | `"bar"` | `"baz"` | $\cdots$	% Strings
$n \in$ `Num`	::=	$0 \mid 1 \mid 2 \mid \cdots$	% Numbers
$b \in$ `Bool`	::=	true | false	% Booleans
$pv \in$ `Prim`	::=	$m \mid n \mid b \mid$ null | undefined	% Primitive Values
$r \in$ `Ref` $\ni$ null			% References
$v \in$ `Val`	::=	$pv \mid r \mid pf$	% Values
$pf \in \mathcal{F}_\lambda$	::=	$\lambda x.\{\text{var } y_1, \cdots, y_n;\ e\}$	% Parsed Function Literals
$fv \in$ `Falsy`	::=	false | 0 | undefined | null	% Falsy Values
$i, j, k \in$ `Index`			% Indexes

Table 2.1: Syntax for Values, Identifiers, and Indexes

named `"xpto"` through its prototype-chain and that this property is bound to a function. In this scenario, when calling `o["xpto"](...)`, the keyword this is bound to the object bound to `o` and not to the object that actually defines `"xpto"` in its prototype-chain. Hence, prototypes can be seen as a device for method sharing in JavaScript. Every function can additionally be called as a *constructor*. However, since in this work we do not model the keyword new, we skip the explanation of this feature and refer the reader to [Flanagan 2011] for a detailed account of the language.

Another important feature of JavaScript is that programs are allowed both to dynamically add new properties to the domain of an object and to delete existing ones. A program can check whether a property is accessible to an object through its prototype-chain using the keyword in. Interestingly, the property look-up construct can also be used to check the existence of properties, since the looking-up of a property that is not defined in the prototype-chain of an object does not yield an error but instead a special value – undefined. Furthermore, the looking-up of a variable that has been declared but has not yet been assigned a value also yields undefined. Besides undefined, JavaScript features another value that is meant to be used as a representation of no value – null. However, in contrast to undefined, the value null is an *assignment value*, meaning that it must be explicitly assigned to a variable/property so that its corresponding look-up yields null.

2.1 Formal Syntax

We define a JavaScript-like language, called Core JavaScript, which is intended to model a realistic subset of the JavaScript specification [3rd edition of ECMA 262 1999]. However, in order to simplify the presentation, we do not model the return statement—functions are assumed to return the value to which their body evaluates. Furthermore, given that most implementations do allow explicit prototype mutation, we depart from [3rd edition of ECMA 262 1999] and include this feature through a special property `"_prot_"`, which Core JavaScript programs can directly manipulate. For instance, `o1["_prot_"] = o2` sets the prototype of the object bound to `o1` to the object bound to `o2`, and `o1["_prot_"]` evaluates to the prototype of the object bound to `o1`.

In Core JavaScript, some expressions are annotated with one or two unique indexes, taken from a set `Index`, for the use of the source-to-source transformations presented in the following chapters. These transformations need to add new, unique identifiers to the program to be

$e, e_0, e_1, e_2 \in \texttt{Expr}$	::=	v	% Value
		$\mid \mathsf{this}^i$	% This
		$\mid x^i$	% Identifier
		$\mid e_0\ \mathsf{op}^i\ e_1$	% Binary operation
		$\mid x = e$	% Variable Assignment
		$\mid e_0[e_1]^i$	% Property Look-up
		$\mid e_0\ \mathsf{in}^i\ e_1$	% Membership Testing
		$\mid e_0[e_1] = e_2$	% Property Assignment
		$\mid \mathsf{delete}^i\ e_0[e_1]$	% Property Deletion
		$\mid e_0(e_1)^i$	% Function Call
		$\mid e_0[e_1](e_2)^i$	% Method Call
		$\mid e_0\ ?^{i,j}\ (e_1) : (e_2)$	% Conditional
		$\mid e_0,\ e_1$	% Sequence
		$\mid \{\,\}^i$	% Object Literal
		$\mid \mathsf{function}^i(x)\{\mathsf{var}\ y_1, \cdots, y_n;\ e\}$	% Function Literal

Table 2.2: Syntax of Expressions

transformed. We make this possible by associating each program construct with one or two unique indexes, which are then used to index a special set of identifiers for the exclusive use of the source-to-source transformations to be presented. We use i, j, and k to represent indexes and we omit the index(es) of an expression whenever they are not needed.

In Core JavaScript, identifiers are taken from a set `Var` ranged over by x, y_1, ..., and y_n and values are taken from a set `Val` ranged over by v. We distinguish three types of values: *primitive values*, taken from a set `Prim` and ranged over by pv, *references*, taken from a set `Ref` and ranged over by r, and *parsed function literals*, taken from a set $\mathcal{F}_\lambda$ and ranged over by pf.

The set `Prim` includes strings, numbers, booleans, as well as the two special values used for the representation of no value: null and undefined. The set `Str` of strings is ranged over by m and p. Typically, we use p for property names and m for arbitrary strings. The set `Num` of numbers is ranged over by n. The set `Bool` of booleans contains two distinct values: false that represents the logical constant *false* and true that represents the logical constant *true*. In JavaScript, some values are coerced to true in contexts where a boolean value is expected, whereas other are coerced to false. The latter are called *falsy values*. The set of all falsy values is denoted by `Falsy` and ranged over by fv. Finally, we use op to represent arbitrary binary operators. References are pointers to objects. However, for convenience, the value null is assumed to be contained in `Ref`. When used in a context where a reference is expected, the value null represents the absence of a reference. Finally, a parsed function literal $pf \in \mathcal{F}_\lambda$ corresponds to the parsed counterpart of a function literal expression (described below).

The formal syntax of values, identifiers, and indexes is given in Table 2.1, whereas the formal syntax of Core JavaScript expressions is given in Table 2.2. The set `Expr` of *expressions* is ranged over by e, e_0, e_1 and e_2 and includes:

- the binary operation e_0 op e_1, that applies the binary operator op to the results of computing e_0 and e_1;
- the variable assignment $x = e$, that sets the value of x to the result of computing e;

- the membership testing expression e_0 in e_1, that evaluates to true if the object to which e_1 evaluates defines the property whose name matches the evaluation of e_0 and evaluates to false otherwise;
- the property look-up $e_0[e_1]$, that evaluates to the value of the property whose name matches the result of the computation of e_1 and is accessible to the object resulting from the computation of e_0 via its prototype-chain;
- the property assignment $e_0[e_1] = e_2$, that sets the value of the property of the object obtained by computing e_0 whose name results from the computation of e_1 to the result of e_2;
- the property deletion delete $e_0[e_1]$, that removes the property whose name results from the computation of e_1 from the domain of the object obtained by computing e_0;
- the function call $e_0(e_1)$, that applies the function that results from computing e_0 to the result of the computation of e_1;
- the method call $e_0[e_1](e_2)$, that applies the method whose name results from computing e_1 and which is accessible to the object obtained from the computation of e_0 via its prototype-chain to the result of the computation of e_2;
- the conditional e_0 ? $(e_1) : (e_2)$, that executes e_1 or e_2 depending on whether the computation of e_0 renders true or false;
- the sequential expression e_0, e_1, that executes e_1 after the execution of e_0 has terminated;
- the object literal expression $\{\}$, that allocates a new object in memory;
- the function literal expression function$(x)\{$var $y_1, \cdots, y_n;\ e\}$, that evaluates to the an *internal object* containing the corresponding parsed function literal;

The set `Expr` of expressions additionally includes values, the this keyword, and identifiers. In the following, we use $e.x$ as an abbreviation for $e[\text{string}(x)]$ (where string(x) denotes the string corresponding to the name of the identifier x) and $e0$? $e1$ as an abbreviation for e_0 ? e_1 : 0.

2.2 Running Example

This section presents the running example that is used throughout the thesis. It consists of a fragment of the code for a simple contact management online application, given in Figure 2.1. The variable `CM` holds the *Contact Manager* object. The contact manager stores contacts in an object that is bound to its property `"contact_list"`. This object is used as a table whose entries are the last names of the contacts (extended with unique integers to avoid collisions) and whose values are the actual contacts. A contact is simply an object containing a first name (stored in property `"fst"`), a last name (stored in property `"lst"`), an e-mail address (stored in property `"email"`), and a flag `"favourite"` (whose existence indicates that that contact is among the user's favourite contacts).

This example illustrates the typical use of prototypical inheritance in JavaScript. We create a "fixed" object for storing all the methods contact objects must implement and we assign this object to the property `"proto_contact"` of the *Contact Manager*. Every time a contact object is created, its prototype is set to `CM.proto_contact`. Hence, every contact object implements the methods: **(1)** `printContact` that generates a string with a description of the contact, **(2)** `makeFavourite` that marks the contact as favourite, **(3)** `isFavourite` that checks whether the

```
CM = { }, CM.proto_contact = { }, CM.contact_list = { },

CM.proto_contact.printContact = function() { this.lst + "," + this."fst" },

CM.proto_contact.makeFavourite = function() { this.favourite = null },

CM.proto_contact.unFavourite = function() {
   "favourite" in this ? delete this["favourite"] : true
},

CM.proto_contact.isFavourite = function() { "favourite" in this },

CM.createContact = function(fst_name, lst_name, email) {
   var contact;
   contact = { },
   contact._prot_ = CM.proto_contact,
   contact.fst = fst_name,
   contact.lst = lst_name,
   contact.email = email,
   contact
},

CM.storeContact = function(contact, i) {
   var list, key;
   list = this.contact_list,
   key = contact.lst + i,
   key in list ? (CM.storeContact(contact, i + 1)) : (list[key] = contact)
},

CM.getContact = function(lst_name, i) { this.contact_list[lst_name + i]}
```

Figure 2.1: A Simple Contact Manager

contact is marked as favourite, and **(4)** `unFavourite` that deletes the property that marks the contact as favourite.

In the following, we give a brief description of the methods that compose the Contact Manager example.

- **Methods of Contact Objects.** The method `printContact` returns a string consisting of the last and first names of the contact on which it was called separated by a comma (in this context, the binary operator + should be interpreted as string concatenation). Since the mere existence of the property `"favourite"` in a contact marks it as a *favourite* contact, the method `makeFavourite` only has to assign an arbitrary value to the property `"favourite"` of a contact to turn that contact into a favourite contact. To stress this fact, we choose to assign it to null. Conversely, in order for a contact to cease to be a favourite contact, one simply has delete the property `"favourite"` from its list of properties. Finally, to check whether a contact is a favourite contact, it suffices to check whether `"favourite"` belongs to its list of properties, which can be done using the program construct in.

- **Methods of the Contact Manager.** The method `createContact` creates a new contact and returns it. Therefore, the last expression in its body is `contact`, since it evaluates to the newly created contact. Given a contact object and an integer n, the method `storeContact` stores the contact corresponding to its first argument in the the the contact

$o \in \texttt{Obj}$	$::=$	$[m_0 \mapsto v_0, m_1 \mapsto v_1, \cdots]$	% Objects
$\mu \in \texttt{Mem}$	$::=$	$[r_0 \mapsto o_0, r_1 \mapsto o_1, \cdots]$	% Memories

Table 2.3: Semantic Domains - Extensional Definitions

list of the contact manager. As mentioned above, a contact list is an object whose entries are the last names of the stored contacts extended with unique integers to avoid collisions. Hence, the method `storeContact` first checks whether there already exists a contact with the same last name associated with n in the contact list. If it is not the case, it stores the contact in the corresponding property of the contact list. If it is the case, the method calls itself recursively with the same contact but with n incremented by one. Finally, the method `getContact` returns the contact associated with the name and integer given as inputs. If no such contact exists, it returns `undefined`.

2.3 Notation

Before proceeding with the description of the formal semantics of Core JavaScript, we must introduce some auxiliary notation for representing sequences of elements and partial mappings, which is then used throughout the thesis.

Sequences In the following, we use $\overrightarrow{z}$ to denote a sequence of elements. Given a sequence $\overrightarrow{z}$ we use: **(1)** $\overrightarrow{z}(i)$ for the i^{th} element of $\overrightarrow{z}$, **(2)** $|\overrightarrow{z}|$ for its number of elements, **(3)** $\mathsf{Shift}_{\mathsf{L}}(\overrightarrow{z}, i)$ for the sequence obtained by removing from $\overrightarrow{z}$ its i^{th} element (provided that it is defined) and left-shifting its remaining elements by one position, **(4)** $\overrightarrow{z} :: z$ for the sequence obtained by appending z to $\overrightarrow{z}$, **(5)** $z :: \overrightarrow{z}$ for the sequence obtained by prepending z to $\overrightarrow{z}$, **(6)** $\overrightarrow{z}_0 :: \overrightarrow{z}_1$ for the concatenation of $\overrightarrow{z}_0$ and $\overrightarrow{z}_1$, and **(7)** $\mathsf{last}(\overrightarrow{z})$ for the last element of $\overrightarrow{z}$. Futhermore, the symbol ε is used to denote the empty sequence.

Partial Mappings We use: **(1)** $[z_0 \mapsto w_0, \cdots, z_n \mapsto w_n]$ for the partial function that maps z_0 to w_0, ..., and z_n to w_n respectively, **(2)** $Z\,[z_0 \mapsto w_0, \cdots, z_n \mapsto w_n]$ for the partial mapping that coincides with Z everywhere except in z_0, ..., and z_n, which are otherwise mapped to w_0, ..., and w_n respectively, **(3)** $Z|_W$ for the restriction of the mapping Z to W (provided that W is included in the domain of Z), **(4)** $Z(z \cdot w)$ for the nested function call $(Z(z))(w)$ (provided that $Z(z)$ is a function), and **(5)** $Z[z \cdot w_1 \mapsto w_0]$ for the nested update $Z\,[z \mapsto Z(z)\,[w_0 \mapsto w_1]]$,

2.4 Formal Semantics

This section describes the formal semantics of Core JavaScript. In Core JavaScript, objects are modelled as unrestricted mappings from strings to values. Hence, an object $o \in \texttt{Obj} : \texttt{Str} \to \texttt{Val}$ is a partial function mapping strings to values. The strings in the domain of an object are called its properties. Given an object o and a property p, the value bound to o's property p is denoted by $o(p)$. Not all properties can be manipulated by Core JavaScript programs. Some properties, called *internal*, are reserved for the use of the semantics, meaning that they can neither be inspected nor updated by JavaScript programs. For clarity, these properties are prefixed with an "@". We use $dom(o)$ to denote the set of properties of o excluding internal properties and $@dom(o)$ for the set of all properties of o including internal properties.

A Core JavaScript memory $\mu \in \texttt{Mem} : \texttt{Ref} \to \texttt{Obj}$ is a partial mapping from references to objects as in [3rd edition of ECMA 262 1999]. Hence, given a memory μ and a reference r, the

object bound to r in μ is denoted by $\mu(r)$. Consequently, given a memory μ, a reference r, and a property p, $(\mu(r))(p)$ denotes the value bound to the property p of the object pointed to by r in μ. Finally, given an object o, we denote by $\#o$ the reference that points to o. Table 2.3 presents the extensional definition of Core JavaScript objects and memories.

As in [Banerjee 2002], we assume a *parametric object allocator*, meaning that references are chosen deterministically. Concretely, the evaluation of an object literal yields a new reference, which is computed using the deterministic allocator fresh, and which is set to point to the newly created object. While allowing us to avoid having to deal with technical details in the stating the security properties (presented in the following chapters), the assumption of a parametric allocator does not weaken the results of the thesis, since, in practice, allocators are in fact deterministic.

2.4.1 Scope Objects

In Core JavaScript, the binding of variables is modelled *in memory* by the use of *scope objects* [Maffeis 2008]. Hence, in the formal semantics, a function/method call triggers the creation of a scope object. A scope object is an internal object that maps the formal parameter of the function that is being called as well as the variables declared in its body to their respective values. A scope object is said to be *active* if it is associated with the function/method that is currently executing. Since function literals can be nested inside each other, every scope object defines a property `"@scope"` that binds the reference of the scope object that was active when the corresponding function literal was evaluated. The sequence of scope objects that can be accessed from a given scope object through the respective `"@scope"` properties is called a *scope-chain*. The *global object*, which is assumed to be pointed to by a fixed reference $\#glob$, is the object that is at the end of every scope-chain and therefore it is the object that binds *global variables*. In particular, we assume that the global object also defines a property `"@scope"`, which is in its case set to null.

In order to determine the value associated with a given variable, one has to inspect all objects in the scope-chain that starts in the *active* scope object. Concretely, when trying to look-up the value bound to an identifier `xpto` in the current scope, the semantics first checks whether $\texttt{"xpto"} \in dom(\mu(r))$, where r is the reference pointing to the current scope object. If $\texttt{"xpto"} \in dom(\mu(r))$, the variable look-up yields $\mu(r \cdot \texttt{"xpto"})$, otherwise the semantics checks whether the next scope object in the current scope-chain defines a binding for `"xpto"`, and so forth. This behaviour is modelled by the semantic function $\mathsf{Scope} : \texttt{Mem} \times \texttt{Ref} \times \texttt{Var} \to \texttt{Ref}$ formally given in Definition 2.1. Informally, $r_1 = \mathsf{Scope}(\mu, r_0, x)$ means that r_1 is the reference that points to the scope object that defines a binding for variable x and that is closest to the one pointed to by r_0 ($\mu(r_0)$) in the scope-chain that starts at $\mu(r_0)$.

Definition 2.1 (Scope)**.** *The function* $\mathsf{Scope} : \texttt{Mem} \times \texttt{Ref} \times \texttt{Str} \to \texttt{Ref}$ *is defined as follows:*

$$\mathsf{Scope}(\mu, r, x) = \begin{cases} \mathsf{null} & \text{if } r = \mathsf{null} \\ r & \text{if } \mathsf{string}(x) \in dom(\mu(r)) \\ \mathsf{Scope}(\mu, \mu(r \cdot \texttt{"@scope"}), x) & \text{otherwise} \end{cases}$$

The variable look-up procedure clearly exposes the duality identifier/string that holds a prominent spot in the formal semantics of Core JavaScript. At runtime, the identifiers declared in the body of a function, as well as its formal parameter, are modelled as properties of a scope object. However, the properties in the domain of an object are strings. Hence, each scope object maps the strings corresponding to the names of the identifiers as well as the formal parameter of its corresponding function to their respective values. Formally, given an identifier x, $\mathsf{string}(x)$ denotes the string corresponding to its name. Conversely, given a string m, $\mathsf{ident}(m)$ denotes the identifier whose name corresponds to m.

2.4.2 Function Objects

In the formal semantics, the evaluation of a function literal yields a reference to an object, called *a function object*, that stores its parsed counterpart. More specifically, since every function is executed in the environment in which the corresponding function literal was evaluated, every function object defines the following two properties:

- `"@code"` that stores the parsed function literal and
- `"@fscope"` that stores the reference that points to the scope object that was active when the corresponding function literal was evaluated.

As an example, assume that the global object defines a variable `out` originally set to `null`. In this scenario, the evaluation of the program presented below on the left yields the value 0 and creates in memory the list of objects displayed below on the right:

```
(function(x){
  var g, h;
  g = function(x) {h(2)},
  h = function(y){out = x},
  g(1)
})(0);
```

$$o_s^0 = [\texttt{"@scope"} \mapsto \#glob, \texttt{"x"} \mapsto 0, \texttt{"g"} \mapsto o_g, \texttt{"h"} \mapsto o_h]$$
$$o_s^g = [\texttt{"@scope"} \mapsto \#o_s^0, \texttt{"x"} \mapsto 1]$$
$$o_s^h = [\texttt{"@scope"} \mapsto \#o_s^0, \texttt{"y"} \mapsto 2]$$
$$o_0 = [\texttt{"@code"} \mapsto \lambda x.\texttt{var g,h;}\ \hat{e}, \texttt{"@fscope"} \mapsto \#glob]$$
$$o_g = [\texttt{"@code"} \mapsto \lambda x.h(2), \texttt{"@fscope"} \mapsto \#o_s^0]$$
$$o_h = [\texttt{"@code"} \mapsto \lambda y.out = x, \texttt{"@fscope"} \mapsto \#o_s^0]$$

where: **(1)** o_s^0, o_s^g, and o_s^h are the scope objects associated with the invocation of the outermost anonymous function, of function g, and of function h, respectively, **(2)** objects o_0, o_g, and o_h are their respective function objects, and **(3)** $\hat{e}$ is the body of the outermost anonymous function. After the execution of this program, the global object maps `out` to 0 and not to 1, because the scope object that is closest to o_s^h and which defines a binding for x is o_s^0 and not o_s^g (which does not belong to the scope-chain of o_s^h).

2.4.3 Scope Allocation

The creation of a scope object is formally emulated by the semantic function $\mathsf{NewScope} : \texttt{Mem} \times \texttt{Ref} \times \texttt{Val} \times \texttt{Ref} \rightarrow \texttt{Mem} \times \texttt{Val} \times \texttt{Ref}$, which is given in Definition 2.2. Intutively, $\langle \mu', e, r' \rangle = \mathsf{NewScope}(\mu, r_f, v_{arg}, r_{this})$ means that r' is the reference of the newly allocated scope object. This scope object is meant to be used as the active scope object during the execution of the function pointed to by r_f in μ. In this new scope object, the formal argument of the function to be executed is bound to the value v_{arg} and the keyword `this` is bound to r_{this}. Finally, the memory that results from the allocation of the scope object is μ' and e is the body of the function to be executed.

Definition 2.2 (NewScope)**.** *For any two memories μ and μ', three references r_f, r_{this}, and r', value v_{arg}, and expression e, $\langle \mu', e, r' \rangle = \mathsf{NewScope}(\mu, r_f, v_{arg}, r_{this})$ holds if and only if:*

- $\lambda x.\{\mathsf{var}\ y_1, \cdots, y_n;\ e\} = \mu(r_f \cdot \texttt{"@code"})$,
- $r = \mu(r_f \cdot \texttt{"@fscope"})$,
- $r' = \mathsf{fresh}()$,
- $\mu' = \mu\,[r' \mapsto [\texttt{"@scope"} \mapsto r, m_x \mapsto v_{arg}, \texttt{"@this"} \mapsto r_{this}, m_{y_1} \mapsto \mathsf{undefined}, \cdots, m_{y_n} \mapsto \mathsf{undefined}]]$, *where:* $m_x = \mathsf{string}(x)$, $m_{y_1} = \mathsf{string}(y_1)$, *...,* *and* $m_{y_n} = \mathsf{string}(y_n)$.

for some identifiers x, y_1, ..., and y_n.

2.4.4 Prototype-Chain Inspection

In Core JavaScript, every object (except scope objects and function objects) defines a property `"_prot_"` that stores a reference pointing to its prototype and which is originally set to null. When trying to look-up the value of a property p of an object o, the semantics first checks whether $p \in dom(o)$. If $p \in dom(o)$, the property look-up yields $o(p)$, otherwise the semantics checks whether the prototype of o (pointed to by $o($`"_prot_"`$)$) defines a property named p, and so forth. The prototype-chain inspection procedure is emulated by the semantic function $\mathsf{Proto} : \mathtt{Mem} \times \mathtt{Ref} \times \mathtt{Str} \to \mathtt{Ref}$ given in Definition 2.3. Informallly, $r' = \mathsf{Proto}(\mu, r, m)$ means that $\mu(r')$ is the object that is closest to $\mu(r)$ in its prototype-chain and that defines a binding for m. Hence, the evaluation of the program:

$$\mathtt{o0} = \{\,\},\ \mathtt{o0.xpto} = 0,\ \mathtt{o1} = \{\,\},\ \mathtt{o1.\_proto\_} = \mathtt{o0},\ \mathtt{o1.xpto} \tag{2.1}$$

yields 0, because, even though the object bound to `o1` does not define the property `"xpto"`, its prototype does.

Definition 2.3 (Proto)**.** *The function* $\mathsf{Proto} : \mathtt{Mem} \times \mathtt{Ref} \times \mathtt{Str} \to \mathtt{Ref}$ *is defined as follows:*

$$\mathsf{Proto}(\mu, r, p) = \begin{cases} \mathsf{null} & \text{if } r = \mathsf{null} \\ r & \text{if } p \in dom(\mu(r)) \\ \mathsf{Proto}(\mu, \mu(r \cdot \texttt{"\_prot\_"}), p) & \text{otherwise} \end{cases}$$

When looking-up the value of a property p in an object o, if p is not defined in the whole prototype-chain of o, instead of yielding an error, the semantics yields undefined. Therefore, the expression $\mathtt{o} = \{\,\}, \mathtt{o.xpto}$ evaluates to undefined.

2.4.5 Method Calls versus Function Calls

A function can be either invoked as a normal function or as a method. When calling a function as a method, the keyword this is bound to the receiver object (that is, the object on which the method was invoked), otherwise it is bound to the global object. Therefore, every scope object defines a property `"@this"` that holds the value of the keyword this in that scope. Hence, suppose that in a memory μ, the global object defines two variables `o0` and `o1` that hold references to the objects $[$`"_prot_"` $\mapsto \mathsf{null}, $`"f"` $\mapsto \#o_f]$ and $[$`"_prot_"` $\mapsto \#o_0]$ respectively, where $\#o_f$ is the reference of a given function object. In the evaluation of the expression `o1.f`(0), the semantics starts by creating a scope object whose property `"@this"` is set to $\#o_1$ and then proceeds with the evaluation of the body of the function pointed to by $\#o_f$.

In contrast to real client-side JavaScript where the global variable `window` holds a reference to the global object, in Core JavaScript a program cannot directly get hold of the reference pointing to the global object. However, any program can obtain this reference by evaluating the expression this in the body of a function called "as a function". For instance, assuming that the global object defines the global variables `x`, `f`, and `global`, after the evaluation of the program:

$$\mathtt{x} = 0,\ \mathtt{f} = \mathsf{function}()\ \{\mathsf{this}\},\ \mathtt{global} = \mathtt{f}(),\ \mathtt{global.x} = 1 \tag{2.2}$$

the global variable `x` is set to 1.

2.4.6 Formal Semantics - Specification

The big-step semantics of Core JavaScript is presented in Figure 2.2. Every big-step semantic transition has the following form: $r \vdash \langle \mu, e \rangle \Downarrow \langle \mu', v \rangle$, where: **(1)** r is the reference of the active scope object, **(2)** μ and μ' are the initial and final memories, **(3)** e is the expression to evaluate, and **(4)** v is the value to which it evaluates. In the following, we give a brief description of each rule:

$$\textsc{Value}\quad r \vdash \langle \mu, v\rangle \Downarrow \langle \mu, v\rangle \qquad \textsc{This}\quad r \vdash \langle \mu, \mathsf{this}\rangle \Downarrow \langle \mu, \mu(r \cdot \texttt{"@this"})\rangle$$

$$\textsc{Variable}\quad \frac{r_x = \mathsf{Scope}(\mu, r, x) \qquad r_x \neq \mathsf{null} \qquad m_x = \mathsf{string}(x)}{r \vdash \langle \mu, x\rangle \Downarrow \langle \mu, \mu(r_x \cdot m_x)\rangle}$$

$$\textsc{Binary Operation}\quad \frac{r \vdash \langle \mu, e_0\rangle \Downarrow \langle \mu_0, v_0\rangle \qquad r \vdash \langle \mu_0, e_1\rangle \Downarrow \langle \mu_1, v_1\rangle \qquad v' = v_0 \ \mathsf{op}\ v_1}{r \vdash \langle \mu, e_0 \ \mathsf{op}\ e_1\rangle \Downarrow \langle \mu_1, v'\rangle}$$

$$\textsc{Variable Assignment}\quad \frac{r \vdash \langle \mu, e\rangle \Downarrow \langle \mu_0, v_0\rangle \qquad r_x = \mathsf{Scope}(\mu_0, r, x) \qquad r_x \neq \mathsf{null} \qquad m_x = \mathsf{string}(x)}{r \vdash \langle \mu, x = e\rangle \Downarrow \langle \mu_0[r_x \cdot m_x \mapsto v_0], v_0\rangle}$$

$$\textsc{Property Look-up}\quad \frac{r \vdash \langle \mu, e_0\rangle \Downarrow \langle \mu_0, r_0\rangle \qquad r \vdash \langle \mu_0, e_1\rangle \Downarrow \langle \mu_1, m_1\rangle \qquad r' = \mathsf{Proto}(\mu_1, r_0, m_1) \qquad r' \neq \mathsf{null} \Rightarrow v = \mu_1(r' \cdot m_1) \qquad r' = \mathsf{null} \Rightarrow v = \mathsf{undefined}}{r \vdash \langle \mu, e_0[e_1]\rangle \Downarrow \langle \mu_1, v\rangle}$$

$$\textsc{Membership Testing}\quad \frac{r \vdash \langle \mu, e_0\rangle \Downarrow \langle \mu_0, m_0\rangle \qquad r \vdash \langle \mu_0, e_1\rangle \Downarrow \langle \mu_1, r_1\rangle \qquad r' = \mathsf{Proto}(\mu_1, r_1, m_0) \qquad r' \neq \mathsf{null} \Rightarrow v = \mathsf{true} \qquad r' = \mathsf{null} \Rightarrow v = \mathsf{false}}{r \vdash \langle \mu, e_0 \ \mathsf{in}\ e_1\rangle \Downarrow \langle \mu_1, v\rangle}$$

$$\textsc{Property Assignment}\quad \frac{r \vdash \langle \mu, e_0\rangle \Downarrow \langle \mu_0, r_0\rangle \qquad r \vdash \langle \mu_0, e_1\rangle \Downarrow \langle \mu_1, m_1\rangle \qquad r \vdash \langle \mu_1, e_2\rangle \Downarrow \langle \mu_2, v_2\rangle}{r \vdash \langle \mu, e_0[e_1] = e_2\rangle \Downarrow \langle \mu_2[r_0 \cdot m_1 \mapsto v_2], v_2\rangle}$$

$$\textsc{Property Deletion}\quad \frac{r \vdash \langle \mu, e_0\rangle \Downarrow \langle \mu_0, r_0\rangle \qquad r \vdash \langle \mu_0, e_1\rangle \Downarrow \langle \mu_1, m_1\rangle \qquad m_1 \neq \texttt{"\_prot\_"} \qquad \mu' = \mu\left[r_0 \mapsto \mu_0(r_0)|_{dom(\mu_0(r_0))\setminus\{m_1\}}\right]}{r \vdash \langle \mu, \mathsf{delete}\ e_0[e_1]\rangle \Downarrow \langle \mu', \mathsf{true}\rangle}$$

$$\textsc{Function Call}\quad \frac{r \vdash \langle \mu, e_0\rangle \Downarrow \langle \mu_0, r_0\rangle \qquad r \vdash \langle \mu_0, e_1\rangle \Downarrow \langle \mu_1, v_1\rangle \qquad \langle \hat{\mu}, \hat{e}, \hat{r}\rangle = \mathsf{NewScope}(\mu_1, r_0, v_1, \#glob) \qquad \hat{r} \vdash \langle \hat{\mu}, \hat{e}\rangle \Downarrow \langle \mu', v\rangle}{r \vdash \langle \mu, e_0(e_1)^i\rangle \Downarrow \langle \mu', v\rangle}$$

$$\textsc{Method Call}\quad \frac{r \vdash \langle \mu, e_0\rangle \Downarrow \langle \mu_0, r_0\rangle \qquad r \vdash \langle \mu_0, e_1\rangle \Downarrow \langle \mu_1, m_1\rangle \qquad r \vdash \langle \mu_1, e_2\rangle \Downarrow \langle \mu_2, v_2\rangle \qquad r_m = \mathsf{Proto}(\mu_2, r_0, m_1) \qquad r_f = \mu_2(r_m \cdot m_1) \qquad \langle \hat{\mu}, \hat{e}, \hat{r}\rangle = \mathsf{NewScope}(\mu_2, r_f, v_2, r_0) \qquad \hat{r} \vdash \langle \hat{\mu}, \hat{e}\rangle \Downarrow \langle \mu', v\rangle}{r \vdash \langle \mu, e_0[e_1](e_2)^i\rangle \Downarrow \langle \mu', v\rangle}$$

$$\textsc{Conditional}\quad \frac{r \vdash \langle \mu, \hat{e}\rangle \Downarrow \langle \hat{\mu}, \hat{v}\rangle \qquad \hat{v} \notin \texttt{Falsy} \Rightarrow i = 0 \qquad \hat{v} \in \texttt{Falsy} \Rightarrow i = 1 \qquad r \vdash \langle \hat{\mu}, e_i\rangle \Downarrow \langle \mu', v\rangle}{r \vdash \langle \mu, \hat{e}\ ?\ (e_0) : (e_1)\rangle \Downarrow \langle \mu', v\rangle}$$

$$\textsc{Sequence}\quad \frac{r \vdash \langle \mu, e_0\rangle \Downarrow \langle \mu_0, v_0\rangle \qquad r \vdash \langle \mu_0, e_1\rangle \Downarrow \langle \mu_1, v_1\rangle}{r \vdash \langle \mu, e_0, e_1\rangle \Downarrow \langle \mu_1, v_1\rangle}$$

$$\textsc{Object Literal}\quad \frac{r' = \mathsf{fresh}() \qquad \mu' = \mu\left[r' \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}]\right]}{r \vdash \langle \mu, \{\ \}^i\rangle \Downarrow \langle \mu', r'\rangle}$$

$$\textsc{Function Literal}\quad \frac{r_f = \mathsf{fresh}() \qquad \mu' = \mu\left[r_f \mapsto [\texttt{"@fscope"} \mapsto r, \texttt{"@code"} \mapsto \lambda x.\{\mathsf{var}\ y_1, \cdots, y_n;\ e\}]\right]}{r \vdash \langle \mu, \mathsf{function}^i(x)\{\mathsf{var}\ y_1, \cdots, y_n;\ e\}\rangle \Downarrow \langle \mu', r'\rangle}$$

Figure 2.2: A Big-Step Semantics for Core JavaScript

- The Rule [Value] simply evaluates a value to itself.
- The Rule [This] evaluates the keyword this to the reference bound to the property `"@this"` of the active scope object.
- The Rule [Variable] starts by looking-up in the current scope-chain the reference of the scope-object that defines a binding for the variable x - r_x. Then, it returns the value with which that scope object associates `"`x`"` (the string that corresponds to the name of x).

- The Rule [Binary Operation] starts by sequentially evaluating the two subexpressions of the current expression, thereby obtaining two values v_0 and v_1. The expression evaluates to the result of applying the corresponding binary operation to v_0 and v_1.
- The Rule [Variable Assignment] starts by evaluating the expression to be assigned, thereby obtaining a value v. This value is then assigned to the property matching the variable to which the value is assigned in the scope object that defines a binding for it.
- The Rule [Property Look-up] starts by sequentially evaluating the two subexpressions of the current expression, thereby obtaining the reference to the object whose property is being inspected (r_0) and the string corresponding to the property's name (m_1). Then, the semantics looks for the object that defines m_1 in the prototype-chain of the object pointed to by r_0. If that object exists, the semantics yields the value with which it associates property m_1. Otherwise, the semantics yields undefined.
- The Rule [Membership Testing] starts by sequentially evaluating the two subexpressions of the current expression, thereby obtaining a reference to an object r_1 and a string m_0. Then the semantics checks whether any of the objects in the prototype-chain of the object pointed to by r_1 defines a property named m_0. If that is the case, the expression evaluates to true. Otherwise, it evaluates to false. It is important to note that the rule [Membership Testing] cannot be simulated by the Rule [Property Look-up], because a property look-up cannot distinguish the case in which an object defines a given property but maps it to the value undefined from the case in which an object does not define a given property.
- The Rule [Property Assignment] starts by sequentially evaluating the three subexpressions of the current expression, thereby obtaining the reference to the object whose property is being updated/created (r_0), the string corresponding to the property's name (m_1), and the value that is to be assigned to it (v_2). Then, the semantics sets the value of the property m_1 in the object pointed to by r_0 to v_2 in the resulting memory. This is done by setting r_0 to point to an object that coincides with $\mu_2(r_0)$ in every property except for m_1, which is set to point to v_2.
- The Rule [Property Deletion] starts by sequentially evaluating the two subexpressions of the current expression, thereby obtaining the reference to the object whose property is to be deleted (r_0) and the string corresponding to the property's name (m_1). Then, the semantics removes m_1 from the domain of the object pointed to by r_0. Note that programs are not allowed to delete the property `"_prot_"` of any given object.
- The Rule [Function Call] starts by sequentially evaluating the two subexpressions of the current expression, thereby obtaining the reference pointing to the function object of the function to be executed (r_0) as well as the value to be used as its argument (v_1). Then, the semantics allocates a new scope object and executes the body of the function in the updated memory. Observe that during the execution of the function's body the keyword this is bound to the reference pointing to the global object.
- The Rule [Method Call] starts by sequentially evaluating the three subexpressions of the current expression, thereby obtaining the reference to the object on which the method is called (r_0), the method's name (m_1), and the value to be used as an argument v_2. Then, the semantics finds the reference pointing to the object in the prototype-chain of the one pointed to by r_0 that actually implements the method named m_1 and obtains the function object corresponding to that method (stored in reference r_f). Finally, the semantics allocates a new scope object and executes the body of the method in the updated memory.

- The Rule [Conditional Expression] starts by evaluating the guard of the conditional expression, thereby obtaining a value – $\hat{v}$. Then, the semantics checks whether $\hat{v}$ is a *falsy* value [Crockford 2008], that is whether $\hat{v} \in \texttt{Falsy} = \{\mathsf{null}, \mathsf{undefined}, \mathsf{false}, 0\}$. If $\hat{v}$ is not a *falsy* value, the then-branch of the conditional is executed. If it is, the else-branch is executed.

- The Rule [Sequence] sequentially evaluates its two subexpressions.

- The Rule [Object Literal] allocates a new object literal in memory. The new object in a new reference and does not have any properties besides `"_prot_"`, which is originally set to null.

- The Rule [Function Literal] allocates a new function object in memory. The property `"@code"` of the new function object stores the parsed counterpart of the corresponding function literal and the property `"@fscope"` stores the reference of the scope object that was active when the function literal was evaluated.

2.5 Related Work

The popularity of JavaScript as a language for developing client-side web applications has been steadily increasing in recent years. This increase in popularity has pushed forward a lot of research in both static and runtime analyses for JavaScript such as: type checking and type inference algorithms [Thiemann 2005, Anderson 2005, Jensen 2009], points-to analysis [Jang 2009], CPS-transformations [Luo 2012, Clements 2008] among others. Most of the analyses for JavaScript in the literature have been designed for different JavaScript-like languages, which capture different aspects of the real language. However, the great majority consists of a core lambda calculus extended with objects supporting prototype-based inheritance and imperative constructs. Some of these works also feature programming constructs for handling exceptions and implicit type coercions [Thiemann 2005].

Maffeis et al. [Maffeis 2008] have been the first to propose a semantics for the full ECMA-262 Standard, 3rd Edition [3rd edition of ECMA 262 1999]. The proposed semantics is small-step and models the binding of variables in memory using scope objects. More recently, Bodin et al. [Bodin 2013] have presented a formalisation of the current version of the ECMA standard [5th edition of ECMA 262 2011] in the Coq proof assistant as well as a JavaScript interpreter that has been proven correct with respect to the authors' specification. Furthermore, they have validated their interpreter using test262, the ECMA conformance test suite. In contrast to [Maffeis 2008], the formal semantics presented in [Bodin 2013] is big-step. This fact allows the authors to closely follow the informal specification, thereby maintaining what they call an *eyeball correspondence* between the standard and its formalisation in the Coq proof assistant. In order to overcome the typical drawbacks of big-step semantics (related to the handling of exceptions and divergence), the authors follow the *pretty big-step* style of Charguéraud [Charguéraud 2013]. Another important difference between these two semantics is that the authors of [Bodin 2013] model scope using *environment records* instead of scope objects. An environment record can be either a *declarative environment record* or an *object environment record*. While declarative environment records provide the local scoping associated with function calls, object environment records provide the dynamic scooping associated with the use of the construct with.

Also with the goal of reasoning precisely about real JavaScript programs, Guah et al. [Guha 2010] have followed, however, a completely different approach from the works mentioned above. They have proposed λ_{JS} – a lambda calculus enriched with some of the most important JavaScript features, such as objects, prototype-based inheritance and constructs for

handling exceptions, which the authors claim to capture the essence of JavaScript. Furthermore, they provide a de-sugaring transformation that compiles arbitrary JavaScript programs into λ_{JS} as well as an interpreter for λ_{JS} programs. These artefacts allowed them to validate their semantics and de-sugaring transformation by testing them against the test262 and Mozilla test suites.

The formal semantics presented in this chapter is heavily inspired by that of Maffeis et al. [Maffeis 2008]. Concretely, it keeps some of its main design features, such as the use of scope objects for the modelling of the binding of variables. However, the size and complexity of the semantics of Maffeis et al. (which occupies more than eighty pages) make it very hard to use it for formally reasoning about the security properties of JavaScript programs. Hence, we opted for the use of a simplified version of this semantics, which retains, in our opinion, the most challenging features of the language in terms of information flow control.

2.6 Discussion

2.6.1 Modelling the Binding of Variables

JavaScript is not **statically scoped** in the sense that, in general, it is not possible to know statically in which scope we can find a property/variable. Consider, for instance, the following JavaScript program:

```
var x, y, obj0, obj1;
x = 0;
obj0 = { };
obj1 = { };
obj1.x = 1;
obj0._proto_ = obj1;
with(obj0){ y = x; }
```
(2.3)

After the execution of this program `y` is assigned to 1 and not to 0, because the construct `with` construct adds the object bound to `obj0` to the front of the current scope-chain, executes the assignment and then restores the scope-chain to its original state. Furthermore, since scope objects are allowed to have prototypes, the scope-chain inspection procedure traverses the prototype-chain of every scope object before going on to the next scope object. However, the current version of the specification [5th edition of ECMA 262 2011] is statically scoped when in *strict mode*, since it does not allow for the use of the most dynamic features of the language, such as the `with` construct.

Since scope objects are assumed not to have a prototype and since we do not include the JavaScript `with` construct, Core JavaScript programs are statically scoped. This means that we could have modelled the binding of variables using substitution, as in other works targeting subsets of the whole language, as [Guha 2010]. However, we have chosen to model scope using scope objects, as in [Maffeis 2008], for two main reasons. First, we envisage to extend the model to deal with a larger subset of the language, which may not be statically scoped. Second, modelling the binding of variables as the binding of properties allows us to simplify the definition of the security property for Core JavaScript, because we can treat variables and properties uniformly.

Chapter 3

Defining Secure Information Flow in Core JavaScript

Contents

This chapter proposes a *noninterference* definition for Core JavaScript, which is in turn used to define what it means for a program to be secure. As a first step toward the definition of noninterference, we show how to label resources in Core JavaScript with security levels. Intuitively, a *security labelling* for a given memory establishes, for each security level, what parts of that memory are visible by an an attacker at that level. This is not easy to define since not only are the contents of the memory visible to an attacker, but also the structure of these contents. We use the term *security policy* for the pair consisting of a lattice of security levels and a security labelling. In the examples, we use the lattice $\mathcal{L} = \{H, L\}$ with $L \sqsubseteq H$ and $H \not\sqsubseteq L$, meaning that resources labelled with L (*low*) are less confidential than those labelled with H (*high*). Hence, H-labelled resources may depend on L-labelled resources, but not the contrary, as that would entail a *security leak*. We use $\sqcap$ and $\sqcup$ for the least upper bound (*lub*) and greatest lower bound (*glb*), respectively. And we use $\bot$ and $\top$ for the *bottom* level and the *top* level, respectively.

3.1 Challenges for IFC in Core JavaScript

This section reviews the main challenges that are raised by the particular features of the language when defining a notion of secure information flow. These challenges are especially relevant to the definition of security labelling for a Core JavaScript memory. Indeed, a security labelling must capture all the possible ways in which an attacker can use the constructs of the language to reveal the contents of a given memory.

Extensible Objects As discussed earlier, in Core JavaScript, the programmer can dynamically add and remove properties from objects. In fact, objects are commonly used as tables whose keys are computed at runtime. Hence, in many contexts, it is not realistic to expect the programmer to statically know the properties of the objects that are created at runtime. However, security-wise, the programmer often knows the security level of the contents of an object

even when its actual properties are not known. For instance, in the Contact Manager example, the precise structure of `contact_list` cannot be statically known because contacts are to be specified dynamically by the user. Nevertheless, the programmer should be allowed to specify a security policy stating, for example, that the e-mail address of every contact in `contact_list` is confidential and therefore of level H.

Leaks via Prototype Mutations The fact that a prototype of an object is allowed to change at runtime may be exploited to encode security leaks. In order to illustrate this, let us return to the example of the Contact Manager (given in Section 2.2). Suppose that the first and last names of a contact are of level L and that we create a new object, bound to `CM.proto_contact_new`, to be used as the prototype of contact objects, that prints contacts in a different way:

$$\texttt{CM.proto\_contact\_new.printContact} = \mathsf{function}()\ \{\mathsf{this}.\texttt{fst} + \texttt{" "} + \mathsf{this}.\texttt{lst}\} \tag{3.1}$$

The output of `printContact` is *low* for the original and new methods, since, in both cases, it only discloses information at level L. However, the expression:

$$\begin{array}{l}\texttt{h} \ ?\ \texttt{c["\_prot\_"]} = \texttt{CM.proto\_contact\_new},\\ \texttt{l} = \texttt{c.printContact}()\end{array} \tag{3.2}$$

encodes an information flow from an H-labelled resource to an L-labelled resource because, depending on the value of the *high* variable `h`, it changes the prototype of the contact bound to `c` and therefore the behaviour of `printContact`, which is supposed to generate a *low* output. Concretely, depending on the value of `h`, the attacker sees the contact printed as either $last\_name, first\_name$ or as $first\_name\ last\_name$. Hence, an information flow control mechanism must be able to detect that the choice of which method to apply in the evaluation of `c.printContact`() effectively depends on H-labelled information.

Leaks via the Checking of the Existence of Properties In Core JavaScript, a program can dynamically add and remove properties from objects. Furthermore, a program can check whether a property is defined in the prototype-chain of an object using the membership testing construct. Thus, the mere existence of a property in the domain of an object may disclose confidential information. For instance, suppose that the user of the contact manager does not want to disclose which are his favourite contacts. In this case, the existence of the property `favourite` in a contact object should be confidential. However, the fact that the value associated with a property is confidential does not imply that its existence is confidential. Suppose that the e-mail addresses of the contents are supposed to be confidential. This does not mean that the existence of the property `email` in a contact should be confidential. In fact, since all contact objects define a property `email` that is not supposed to be deleted, the existence of that property does not reveal any confidential information.

Leaks via the Global Object In Core JavaScript functions can be invoked using function calls or method calls. During the execution of a method call, the keyword this is bound to the object on which the method was invoked. However, during the execution of a function call, the keyword this is bound to the global object (the object whose properties are the global variables of the program). Hence, it is possible to encode insecure information flows regarding **confidential global variables** using the keyword this inside a function. For instance, the program $\mathsf{function}()\ \{\texttt{l} = \mathsf{this}.\texttt{cookie}\}()$ produces the same effect as `l = cookie`. Dynamic information flow control mechanisms are able to prevent this type of leak very simply, since it amounts to check whether the keyword this is bound to the global object. In contrast, static mechanisms for information flow control face a much more difficult challenge, since it is very

difficult to determine statically whether the keyword this may be bound to the global object in a given program point.

3.2 The Attacker Model

In order to formally characterize the observational power of an attacker, we define a notion of *low-projection* of a memory at a given security level σ [Almeida Matos 2009]. The low-projection of a memory at a given security level σ corresponds to the part of that memory that an attacker at level σ can see.

We start by formally defining a security labelling as a tuple $\Sigma = \langle \Sigma_0, \Sigma_1, \Sigma_2 \rangle$ composed of three partial functions $\Sigma_0 : \texttt{Ref} \mapsto \mathcal{L}$, $\Sigma_1 : \texttt{Ref} \mapsto \texttt{Str} \mapsto \mathcal{L}$, and $\Sigma_2 : \texttt{Ref} \mapsto \texttt{Str} \mapsto \mathcal{L}$ respectively called *object labelling*, *property-value labelling*, and *property-existence labelling* and described below.

- The *object labelling* Σ_0 maps each reference in its domain to the security level associated with the object to which it points, called *object level*. Intuitively, the fact that an object has a visible *object level* means that its existence is observable.
- The *property-value labelling* Σ_1 maps each pair in its domain consisting of a reference and a property name to the *value level* of that property in the object pointed to by that reference. Intuitively, the fact that the property p of the object pointed to by reference r has a visible *value level* means that the value associated with that property in that object is observable.
- The *property-existence labelling* Σ_2 maps each pair in its domain consisting of a reference and a property name to the *existence level* [Hedin 2012] of that property in the object pointed to by that reference. Intuitively, the fact that the property p of the object pointed to by reference r has a visible *existence level* means that the existence of that property in that object is observable.

In the following, we use Lab to denote the set of security labellings. Given a labelling Σ, we denote by $\Sigma.\mathsf{obj}$, $\Sigma.\mathsf{val}$, and $\Sigma.\mathsf{exist}$ the corresponding object labelling, property-value labelling, and property-existence labelling. Consequently, given an object o pointed to by a reference r, a labelling Σ, and a property name p: **(1)** $\Sigma.\mathsf{obj}(r)$ is the object level of o, **(2)** $\Sigma.\mathsf{val}(r \cdot p)$ is the value level of o's property p, and **(3)** $\Sigma.\mathsf{exist}(r \cdot p)$ is the existence level of o's property p. Since not all program resources need to be labelled, a security labelling may be *partial*. However, there are some criteria it must verify. Namely, we say that memory μ is well-labelled by Σ if: $dom(\Sigma.\mathsf{obj}) = dom(\Sigma.\mathsf{val}) = dom(\Sigma.\mathsf{exist}) \subseteq dom(\mu)$ **and** for every reference $r \in dom(\Sigma.\mathsf{obj})$, $@dom(\Sigma.\mathsf{val}(r)) = @dom(\Sigma.\mathsf{exist}(r)) \subseteq @dom(\mu(r))$.

Definition 3.1 formalises the notions of *low-projection* and of *low-equality*. Informally, given a security labelling Σ, an attacker at level σ can see:

- the existence of the objects whose object levels are $\sqsubseteq \sigma$,
- the existence of properties in visible objects whose existence levels are $\sqsubseteq \sigma$,
- the values associated with visible properties in visible objects whose value levels are $\sqsubseteq \sigma$.

Definition 3.1 (Low-Projection and Low-Equality for Core JavaScript Memories)**.** *The low-projection of a memory μ w.r.t. a security level σ and a labeling Σ is given by:*

$$
\begin{aligned}
\mu \restriction^{\Sigma,\sigma} = \ & \{(r, \Sigma.\mathsf{obj}(r)) \mid \Sigma.\mathsf{obj}(r) \sqsubseteq \sigma\} \\
& \cup \{(r, p, \Sigma.\mathsf{exist}(r \cdot p)) \mid \ \Sigma.\mathsf{obj}(r) \sqcup \Sigma.\mathsf{exist}(r \cdot p) \sqsubseteq \sigma \ \wedge \ p \in @dom(\mu(r))\} \\
& \cup \{(r, p, v, \Sigma.\mathsf{val}(r \cdot p)) \mid \ \Sigma.\mathsf{obj}(r) \sqcup \Sigma.\mathsf{exist}(r \cdot p) \sqcup \Sigma.\mathsf{val}(r \cdot p) \sqsubseteq \sigma \ \wedge \ p \in @dom(\mu(r))\}
\end{aligned}
$$

Two memories μ_0 and μ_1, respectively labelled by Σ_0 and Σ_1 are said to be low-equal at security level σ, written $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$ if they coincide in their respective low-projections, $\mu_0 \restriction^{\Sigma_0,\sigma} = \mu_1 \restriction^{\Sigma_1,\sigma}$.

Returning to the Contact Manager example, suppose the user wants to enforce a security policy such that only the e-mails of the stored contacts and the identity of the *favourite* contacts should be of level H. Everything else should be set to L. Figure 3.1 presents the memory resulting from the execution of the program below:

$$\begin{array}{l} \texttt{x} = \texttt{CM.createContact}(\texttt{"Jane"}, \texttt{"Doe"}, \texttt{"jane.d@gmail.com"}), \\ \texttt{y} = \texttt{CM.createContact}(\texttt{"John"}, \texttt{"Doe"}, \texttt{"john.d@gmail.com"}), \\ \texttt{CM.storeContact}(\texttt{x}, 0), \\ \texttt{CM.storeContact}(\texttt{y}, 0), \\ \texttt{makeFavorite}(x) \end{array} \tag{3.3}$$

together with its low-projection at level L. Remark that, while the values of both e-mail addresses are hidden, their existence remains visible. In contrast, the property `favourite` is removed from the contact object of Jane.

3.2.1 Low-Equality for Values and Sequences of Values

In order to ease the presentation of the results of the following chapters, we define a notion of low-equality for *labelled values* and for labelled sequences of values. A labelled value is simply a pair consisting of a value and a security level. Analogously, a labelled sequence of values is a sequence of values paired up with a sequence of security levels. Each level in the sequence of levels labels that occupies the same position in the sequence of values.

Informally, two values v_0 and v_1 respectively labelled by σ_0 and σ_1 are said to be low-equal at level σ, written $v_0, \sigma_0 \sim_\sigma v_1, \sigma_1$, if either they are both observable and coincide or they are both unobservable. This notion is formalised in Definition 3.2.

Definition 3.2 (Low-Equality for Labelled Values)**.** *Two values v_0 and v_1 respectively labelled by the security levels σ_0 and σ_1 are low-equal at a security level σ, written $v_0, \sigma_0 \sim_\sigma v_1, \sigma_1$, if and only if it holds that: $v_0 = v_1 \ \wedge \ \sigma_0 = \sigma_1 \sqsubseteq \sigma \ \vee \ \sigma_0 \sqcap \sigma_1 \not\sqsubseteq \sigma$.*[1]

Definition 3.3 extends the definition of low-equality for labelled values to sequences of labelled values. Informally, two sequences of labelled values are low-equal at a given security level if they are low-equal point-wise. Furthermore, if two sequences of labelled values are low-equal at a given security level, either they have the same number of elements, or the extra elements of the sequence with more elements are not observable.

Definition 3.3 (Low-Equality for Sequences)**.** *Two sequences of values $\overrightarrow{v}_0$ and $\overrightarrow{v}_1$ respectively labelled by two sequences of security levels $\overrightarrow{\sigma}_0$ and $\overrightarrow{\sigma}_1$ are said to be low-equal with respect to a security level σ, written $\overrightarrow{v}_0, \overrightarrow{\sigma}_0 \sim_\sigma \overrightarrow{v}_1, \overrightarrow{\sigma}_1$ if the following hold:*

- $\forall_{0 \leq i < n} \ \overrightarrow{\sigma}_0(i) \sqcap \overrightarrow{\sigma}_1(i) \sqsubseteq \sigma \Rightarrow \overrightarrow{v}_0(i) = \overrightarrow{v}_1(i) \ \wedge \ \overrightarrow{\sigma}_0(i) = \overrightarrow{\sigma}_1(i) \sqsubseteq \sigma$,
- $\forall_{n < i < |\overrightarrow{v}_0|} \ \overrightarrow{\sigma}_0(i) \not\sqsubseteq \sigma$, *and*
- $\forall_{n < j < |\overrightarrow{v}_1|} \ \overrightarrow{\sigma}_1(j) \not\sqsubseteq \sigma$

where $n = \mathsf{min}(|\overrightarrow{v}_0|, |\overrightarrow{v}_1|)$.

[1] This formula can be equivalently re-written as $(\sigma_0 \sqsubseteq \sigma \ \vee \ \sigma_1 \sqsubseteq \sigma) \Rightarrow (\sigma_0 = \sigma_1 \sqsubseteq \sigma \ \wedge \ v_0 = v_1)$.

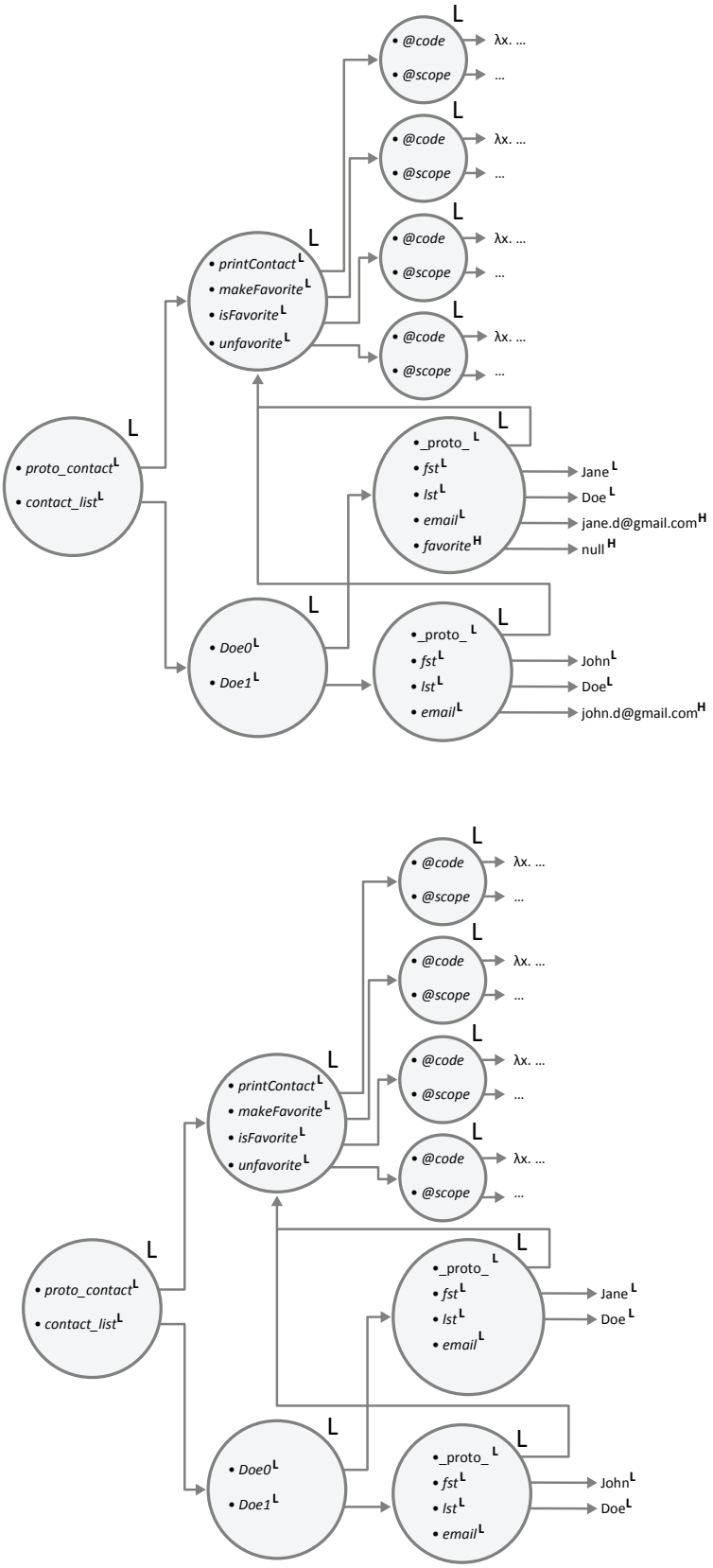


Figure 3.1: A labelled memory and its low-projection

3.3 Noninterferent Allocator

Throughout the thesis, we always assume that object allocators are deterministic. However, it must be possible to relate references created in *low* contexts in different executions. To this end, the definition of object allocator must be modified so that the allocator is given additional information. A parametric object allocator is now defined as a function $\mathsf{fresh} : \mathcal{L} \rightarrow \mathtt{Ref}$ that receives as input a security level σ and outputs a new reference. The security level given as input to the object allocator is referred to as the *level of the allocation*. Intuitively, when an allocation takes place in a *high* context, it is given a *high* security level. And, when an allocation is performed in a *low* context, it is given a *low* security level.

We do not give the specification of a concrete object allocator. Moreover, we assume that the object allocator has an internal state for recalling how many times it was invoked at each security level (and which were the references generated by the allocation). Throughout the thesis, we assume that allocators are such that: the allocation at level σ of an object in two memories that are low-equal at level σ yields the same reference.

3.4 Related Work

Since the seminal works of Bell and La Padula and Denning [Bell 1976, Denning 1976], the classical approach to secure information flow is to use a lattice of secure levels and a security labelling that maps resources to security levels. The ordering relation on the security levels establishes which are the legal information flows. Information is allowed to move up in the security lattice (from *low*-labelled resources to *high*-labelled resources), but not down. This property was first formally stated via a notion *strong dependency* by Cohen in [Cohen 1977], and later referred to as *noninterference* by Goguen and Messeguer in [Goguen 1982].

In general, one can view *noninterference* as a class of properties that state how the execution of a program is allowed to propagate dependencies between the resources on which it operates. In order to instantiate noninterference to a concrete programming language, one must start by defining how to label program states. While simple imperative languages only require a very simple labelling strategy [Volpano 1996], more complex languages may require sophisticated labelling strategies whose details heavily depend on the features of the targeted language [Banerjee 2002].

Hedin *et al* [Hedin 2012] have been the first to propose an information flow monitor for a realistic core of JavaScript. They introduce the notion of *existence levels* to deal with the constructs for the checking of the existence of properties. They further introduce the notion of *structure security level* (SSL), which corresponds to an upper bound on the existence levels of the properties of an object. Hence, if an object o has a *low* SSL, one can only change its structure (either by adding properties to o or removing properties from o) in low contexts.

3.5 Discussion

3.5.1 Toward an Attacker Model for the ECMA Standard

The attacker model we present here fits the expressiveness of Core JavaScript. The Ecma standard [5th edition of ECMA 262 2011], however, allows for other types of attacks. Namely, in JavaScript, an attacker can explore time-based covert channels [Agat 2000] to encode illegal information flows, which is not the case in Core JavaScript. Consider, for instance, the program

below:

```
l1 = (new Date()).getTime(),
if (h){
    //do meaningless time-consuming operations                    (3.4)
}
l2 = (newDate()).getTime() − l1
```

where the expression `new Date()` evaluates to an object that represents the current date, which, in turn, implements a method `getTime` that outputs the time in milliseconds since 1970/01/01. After the execution of this program, the value of `l2` depends on the initial value of the *high* variable `h`. Therefore, information flow control mechanisms targeting the full Ecma standard must be able to detect these types of flows.

3.5.2 Further Remarks on the Structure Security Level

It is important to emphasise that the *structure security level* [Hedin 2012] is not a key element for the characterisation of the attacker model inherent to JavaScript, but rather a device of the authors' enforcement mechanism. The need for the SSL arises from the fact that the existence levels are not established *a priori*. Hence, the SSL plays the role of the existence level of the properties that are not associated with an existence level. Accordingly, the level associated with the look-up of a property that does not have an existence level is the SSL. Consider the following example:

```
o = { },
h ? (o.xpto = 0),                                               (3.5)
l = "xpto" in o
```

This program encodes an implicit flow from the *high* variable `h` to the *low* variable `l` via the existence of property `"xpto"` in the object bound to `o`. Now suppose we want to design a dynamic mechanism for enforcing secure information flow in Core JavaScript. When executing this program starting from a memory that maps `h` to 1, this implicit flow can be easily detected, since the existence level of property `"xpto"` is H (as it was created in a *high* context). However, when `h` is initially set to 0, it becomes impossible for a dynamic mechanism to identify the implicit flow via the existence level of property `"xpto"`, simply because **it does not exist**. In order to solve this problem, Hedin *et al* have introduced the notion of *structure security level*. The idea is to use this level as the existence level of the properties that do not exist. This means that the structure security level establishes an upper bound on the levels of the contexts in which one is allowed to change the domain of an object (either by adding new properties or removing existing ones). However, as shown here, this level is not needed to characterise the observational power of an attacker in Core JavaScript, but it is rather a design strategy used by dynamic enforcement mechanisms.

Chapter 4

Dynamic Information Flow Control in Core JavaScript

Contents

As JavaScript is a highly dynamic language, it comes as expected that research efforts directed towards defining mechanisms that would check the noninterference of JavaScript programs predominantly feature dynamic approaches, such as information flow monitors [Austin 2012, Hedin 2012] and secure multi-execution [Devriese 2010]. In practice, there are two main ways one could implement a JavaScript information flow monitor: either one modifies a JavaScript engine so that it additionally implements the security monitor (as in [Hedin 2012]), or one inlines the monitor into the original program (as in [Magazinius 2012, Chudnov 2010]). We have chosen to follow the second approach, which has the advantage of being *browser-independent*.

This chapter presents a compiler that inlines an information flow monitor for Core JavaScript. The proposed compiler is proven sound with respect to a standard definition of input-output termination insensitive noninterference for monitors. Informally, we prove that the execution of a compiled program goes through only if that execuction is secure; otherwise, the constraints inlined in the program by the compiler will cause it to diverge.

More specifically, we start by presenting an information flow monitored semantics for Core JavaScript that is proven *sound*, i.e. proven to enforce termination-insensitive noninterference. The proposed monitored semantics differs from a previous monitor for enforcing secure information flow in a realistic core of JavaScript [Hedin 2012] in that it was specifically designed to serve as guide for the implementation of an inlining compiler, rather than for a browser instrumentation. Then, we present an inlining compiler that rewrites Core JavaScript programs in order to simulate their execution in the monitor. The compiler is proven *correct*, meaning that the execution of a program goes through in the monitor *if and only if* the execution of its instrumentation by the inlining compiler goes through in the original semantics. In order for this to be achieved, security labelling is instrumented in the memory of the program, giving rise

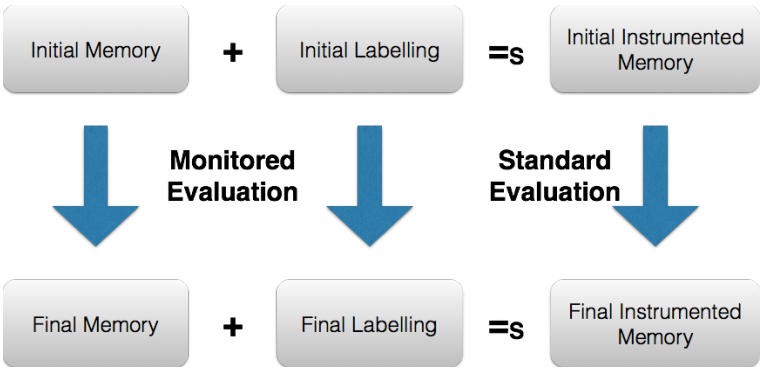


Figure 4.1: Monitored Execution of Program vs. Unmonitored Execution of Compilation

to a *similarity relation* between *labelled memories* and *instrumented memories*. As illustrated in Figure 4.1, given a labelled memory and its instrumented counterpart, the monitored execution of the original program in the labelled memory and the standard execution of its compilation in the instrumented memory always yield two memories that are similar. We have implemented a prototype of the proposed compiler, which supports a subset of JavaScript semantics larger than the one modelled in Core JavaScript and which is available at [Fragoso Santos 2014].

Outline. This chapter is structured as follows: In Section 4.1, we present an information flow monitored semantics for Core JavaScript that is proven *sound*, i.e. proven to enforce termination-insensitive noninterference. Section 4.2 features an inlining compiler that rewrites Core JavaScript programs in order to simulate their execution in the monitor. In Section 4.3 we provide a discussion on related work, whereas in Section 4.4, we elaborate on certain details regarding the implementation of the compiler.

4.1 Monitoring Secure Information Flow in Core JavaScript

In this section, we present a monitored semantics for dynamically enforcing secure information flow in Core JavaScript. The security monitor we present is flow-sensitive, purely dynamic and follows the *no-sensitive-upgrade* discipline of Zdancewic [Zdancewic 2002, Austin 2009].

The monitored execution of an expression e in a memory μ paired up with a security labelling Σ can be interpreted as an extension of the unmonitored execution of e in μ that additionally performs the *abstract execution* of e in Σ [Hunt 2006]. In this sense, we can view Σ as an abstract memory. While the standard execution of e in μ produces a value, its abstract execution in Σ generates a security level σ, which is called the *reading effect* of e [Sabelfeld 2003a]. The reading effect of e corresponds to the least upper bound on the levels of the resources on which the value to which e evaluates depends. The rules of the monitored semantic relation, $\Downarrow_{IF}$, are defined in Figures 4.3 and 4.4. The semantic rules have the form $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle$, where:

- σ_{pc} is *the security level of the program counter*, that is, the security level of the current execution context,
- Σ and Σ' are the initial and final security labellings, and
- σ is the *reading effect* of e.

All the remaining elements keep their original meaning. For simplicity, the monitor was designed in such a way that the reading effect of an expression is always higher than or equal to the level

of the context in which it was evaluated. For clarity, in the specification of each semantic rule, we use:

- light grey for the parts of the rule that coincide with the unmonitored semantics,
- orange for the labelling updates,
- red for the constraints.

The security level associated with checking the existence of a given property in a given object is the corresponding *existence level*. It is natural for a dynamic enforcement mechanism to set the existence level of a given property to the level of the context in which it was created. This, however, raises the problem of deciding which is the existence level of the properties that do not exist yet. For instance, suppose a program checks whether an object o defines a given property p. If p is in the domain of o, the security level of the result should be the existence level of p. But what if p is not in the domain of o? To cope with this issue, each object is associated with a *default existence level* that acts as the existence level of the properties that do not exist yet and which is called *structure security level* [Hedin 2012]. Hence, in the previous example, when p is not in the domain of o, the level of the result should be the structure security level of o.

To keep track of the structure security levels of the objects in memory, dynamic security labellings are extended with a fourth element, called the *structure security labelling*, which maps each object reference to the structure security level of the corresponding object. Furthermore, we assume that object literals are annotated with their corresponding structure security levels. Given a security labelling Σ, we denote the corresponding structure security labelling by $\Sigma.\mathsf{struct}$.

In order to simplify the specification of the monitor, we introduce a group of functions to update security labellings, which are presented in Figure 4.2 and are briefly described below:

- $\mathsf{updt}(\Sigma, (r, p), (\sigma, \sigma'))$ outputs the security labelling obtained from Σ by setting the value level of the property p in the object pointed to by r to σ'. Furthermore, if this object does not define p, its existence level is set to σ.
- $\mathsf{contract}(\Sigma, r, p)$ outputs the security labelling obtained from Σ by removing the existence level and the value level of the property p in the object pointed to by r.
- $\mathsf{extend}(\Sigma, r, \sigma_o, \sigma_s)$ outputs the security labelling obtained from Σ when allocating a new object in the reference r with object level σ_o and structure security level σ_s.

In the following we give a brief description of the rules of the monitored semantics. We ignore by now some important aspects of the monitor, such as the constraints that it enforces, which are carefully discussed in the following subsections. As a general remark, if a rule does not change the memory, it also does not change the security labelling.

- [Value] The reading effect of a value is simply the level of the program counter.
- [This] The reading effect of the expression this is the *lub* between the level of the program counter and the *value level* of the internal property `"@this"` of the current scope object.
- [Variable] The reading effect of a variable x is the *lub* between the level of the program counter and the *value level* of the property m_x of the scope object that defines a binding for x in the current scope-chain (where $m_x = \mathsf{string}(x)$).
- [Binary Operation] The reading effect of a binary operation e_0 op e_1 is simply the *lub* between the reading effects of e_0 and e_1. It is important to emphasise that both the reading effect of e_0 and the reading effect of e_1 are already higher than or equal to the level of the program counter. Hence, the reading effect of e_0 op e_1 is also higher than or equal to the level of the program counter.

$$\textsc{Labelling Update}$$
$$\frac{\begin{array}{c} p \in dom(\Sigma.\mathsf{exist}(r)) \Rightarrow \Sigma_{exist} = \Sigma.\mathsf{exist} \\ p \notin dom(\Sigma.\mathsf{exist}(r)) \Rightarrow \Sigma_{exist} = \Sigma.\mathsf{exist}[r \cdot p \mapsto \sigma] \\ \Sigma_{val} = \Sigma.\mathsf{val}[r \cdot p \mapsto \sigma'] \end{array}}{\mathsf{updt}(\Sigma, (r,p), (\sigma, \sigma')) = \langle \Sigma.\mathsf{obj}, \Sigma_{val}, \Sigma_{exist}, \Sigma.\mathsf{struct} \rangle}$$

$$\textsc{Labelling Contraction}$$
$$\frac{P = @dom(\Sigma.\mathsf{exist}(r)) \backslash \{p\} \qquad \Sigma_{val} = \Sigma.\mathsf{val}\,[r \mapsto \Sigma.\mathsf{val}(r)|_P] \qquad \Sigma_{exist} = \Sigma.\mathsf{exist}\,[r \mapsto \Sigma.\mathsf{exist}(r)|_P]}{\mathsf{contract}(\Sigma, r, p) = \langle \Sigma.\mathsf{obj}, \Sigma_{val}, \Sigma_{exist}, \Sigma.\mathsf{struct} \rangle}$$

$$\textsc{Labelling Extension}$$
$$\frac{\Sigma_{obj} = \Sigma.\mathsf{obj}\,[r \mapsto \sigma_o] \qquad \Sigma_{val} = \Sigma.\mathsf{val}\,[r \mapsto [\,]] \qquad \Sigma_{exist} = \Sigma.\mathsf{exist}\,[r \mapsto [\,]] \qquad \Sigma_{struct} = \Sigma.\mathsf{struct}\,[r \mapsto \sigma_s]}{\mathsf{extend}(\Sigma, r, \sigma_o, \sigma_s) = \langle \Sigma_{obj}, \Sigma_{val}, \Sigma_{exist}, \Sigma_{struct} \rangle}$$

Figure 4.2: Meta-Functions for Updating Security Labellings

- [Variable Assignment] The reading effect of a variable assignment $x = e_0$ is simply the reading effect of e_0, which is already higher than or equal to the level of the program counter. This rule also sets the *value level* of the property m_x of the scope object that defines a binding for x in the current scope-chain to the reading effect of e_0 (where $m_x = \mathsf{string}(x)$). The *existence level* of m_x in that scope-object remains unchanged, because m_x already exists in the scope object that defines a binding for it. The constraint of this rule, as all the other constraints, is explained in Subsection 4.1.1.

- [Property Look-up] The reading effect of a property look-up $e_0[e_1]$ is the *lub* between: **(1)** the reading effects of e_0 and e_1, **(2)** the level of the prototype-chain inspection procedure (explained in Subsection 4.1.3), and **(3)** the *value level* of the property m_1 (obtained from the evaluation of e_1) of the object that defines a binding for it in the prototype-chain of the object pointed to by r_0 (obtained from the evaluation of e_0), **provided that such an object exists.**

- [Membership Testing] The reading effect of a membership testing expression e_0 in e_1 is the *lub* between: **(1)** the reading effects of e_0 and e_1, **(2)** the level of the prototype-chain inspection procedure (explained in Subsection 4.1.3), and the *existence level* of the property m_0 (obtained from the evaluation of e_0) of the object that defines a binding for it in the prototype-chain of the object pointed to by r_1 (obtained from the evaluation of e_1), **provided that such an object exists.**

- [Property Assignment] The reading effect of a property assignment $e_0[e_1] = e_2$ is simply the reading effect of e_2. This rule also sets the *value level* of property m_1 (obtained from the evaluation of e_1) of the object pointed to by r_0 (obtained from the evaluation of e_0) to the *lub* between the reading effects of the **three** subexpressions. If the property assignment is a property creation (meaning that m_1 is not already defined by the object pointed to by r_0), the existence level of m_1 in the object pointed to by r_0 is set to the *lub* between the reading effects of e_0 and e_1.

- [Property Deletion] The reading effect of a property deletion delete $e_0[e_1]$ is simply the level of the program counter, as a property deletion does not reveal any information about

VALUE

$$r, \sigma_{pc} \vdash \langle \mu, v, \Sigma \rangle \Downarrow_{IF} \langle \mu, v, \Sigma, \sigma_{pc} \rangle$$

THIS

$$\frac{r_{this} = \mu(r \cdot \texttt{"@this"}) \qquad \sigma_{this} = \Sigma.\mathsf{val}(r \cdot \texttt{"@this"}) \sqcup \sigma_{pc}}{r, \sigma_{pc} \vdash \langle \mu, \mathsf{this}, \Sigma \rangle \Downarrow_{IF} \langle \mu, r_{this}, \Sigma, \sigma_{this} \rangle}$$

VARIABLE

$$\frac{r_x = \mathsf{Scope}(\mu, r, x) \qquad r_x \neq \mathsf{null} \qquad m_x = \mathsf{string}(x) \qquad \sigma = \Sigma.\mathsf{val}(r_x \cdot m_x) \sqcup \sigma_{pc}}{r, \sigma_{pc} \vdash \langle \mu, x, \Sigma \rangle \Downarrow_{IF} \langle \mu, \mu(r_x \cdot m_x), \Sigma, \sigma \rangle}$$

BINARY OPERATION

$$\frac{\forall_{i=0,1}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \rangle \qquad v' = v_0\ \mathsf{op}\ v_1 \qquad \sigma' = \sigma_0 \sqcup \sigma_1}{r, \sigma_{pc} \vdash \langle \mu_0, e_0\ \mathsf{op}\ e_1, \Sigma_0 \rangle \Downarrow_{IF} \langle \mu_2, v', \Sigma_2, \sigma' \rangle}$$

VARIABLE ASSIGNMENT

$$\frac{r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, v_0, \Sigma_0, \sigma_0 \rangle \qquad r_x = \mathsf{Scope}(\mu_0, r, x) \qquad r_x \neq \mathsf{null} \qquad m_x = \mathsf{string}(x) \qquad \mu' = \mu_0[r_x \cdot m_x \mapsto v_0] \qquad \Sigma' = \mathsf{updt}(\Sigma_0, (r_x, m_x), (\Sigma_0.\mathsf{exist}(r_x \cdot m_x), \sigma_0)) \qquad \sigma_{pc} \sqsubseteq \Sigma_0.\mathsf{val}(r_x \cdot m_x)}{r, \sigma_{pc} \vdash \langle \mu, x = e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu', v_0, \Sigma', \sigma_0 \rangle}$$

PROPERTY LOOK-UP

$$\frac{\begin{array}{c}\forall_{i=0,1}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \rangle \\ \langle r', \sigma' \rangle = \mathsf{Proto}(\mu_2, v_0, v_1, \Sigma_2) \qquad \sigma'' = \sigma_0 \sqcup \sigma_1 \sqcup \sigma' \\ r' = \mathsf{null} \Rightarrow v = \mathsf{undefined} \wedge \sigma = \sigma'' \\ r' \neq \mathsf{null} \Rightarrow v = \mu_1(r' \cdot m_1) \wedge \sigma = \sigma'' \sqcup \Sigma.\mathsf{val}(r' \cdot v_1)\end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, e_0[e_1], \Sigma \rangle \Downarrow_{IF} \langle \mu_2, v, \Sigma_2, \sigma \rangle}$$

MEMBERSHIP TESTING

$$\frac{\begin{array}{c}\forall_{i=0,1}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \rangle \\ \langle r', \sigma' \rangle = \mathsf{Proto}(\mu_2, v_0, v_1, \Sigma_2) \qquad \sigma = \sigma_0 \sqcup \sigma_1 \sqcup \sigma' \\ r' = \mathsf{null} \Rightarrow v = \mathsf{false} \\ r' \neq \mathsf{null} \Rightarrow v = \mathsf{true}\end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, e_0\ \mathsf{in}\ e_1, \Sigma \rangle \Downarrow_{IF} \langle \mu_2, v, \Sigma_2, \sigma \rangle}$$

PROPERTY ASSIGNMENT

$$\frac{\begin{array}{c}\forall_{i=0,1,2}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \rangle \qquad v_0 \in \texttt{Ref} \qquad v_1 \in \texttt{Str} \\ \mu' = \mu_3[v_0 \cdot v_1 \mapsto v_2] \qquad \Sigma' = \mathsf{updt}(\Sigma_3, (v_0, v_1), (\sigma_0 \sqcup \sigma_1, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2)) \\ v_1 \in dom(\mu_3(v_0)) \Rightarrow \sigma_0 \sqcup \sigma_1 \sqsubseteq \Sigma_3.\mathsf{val}(v_0 \cdot v_1) \qquad v_1 \notin dom(\mu_3(v_0)) \Rightarrow \sigma_0 \sqcup \sigma_1 \sqsubseteq \Sigma_3.\mathsf{struct}(v_0)\end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, e_0[e_1] = e_2, \Sigma \rangle \Downarrow_{IF} \langle \mu', v_2, \Sigma', \sigma_2 \rangle}$$

PROPERTY DELETION

$$\frac{\begin{array}{c}\forall_{i=0,1}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \rangle \\ v_0 \in \texttt{Ref} \qquad v_1 \in \texttt{Str} \qquad v_0 \in dom(\mu_2(v_1)) \\ \mu' = \mu_2\left[v_0 \mapsto \mu_2(v_0)|_{dom(\mu_2(v_0)) \setminus v_1}\right] \\ \Sigma' = \mathsf{contract}(\Sigma_2, v_0, v_1) \qquad \sigma_0 \sqcup \sigma_1 \sqsubseteq \Sigma_0.\mathsf{exist}(v_0 \cdot m_1)\end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, \mathsf{delete}\ e_0[e_1], \Sigma_0 \rangle \Downarrow_{IF} \langle \mu', \mathsf{true}, \Sigma', \sigma_{pc} \rangle}$$

OBJECT LITERAL

$$\frac{\begin{array}{c}r_o = \mathsf{fresh}(\sigma_{pc}) \qquad \sigma = \sigma_s \sqcup \sigma_{pc} \\ \mu' = \mu\left[r_o \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}]\right] \\ \Sigma' = \mathsf{extend}(\Sigma, r_o, \sigma_{pc}, \sigma) \\ \Sigma'' = \mathsf{updt}(\Sigma', (r_o, \texttt{"\_prot\_"}), (\sigma, \sigma))\end{array}}{r, \sigma_{pc} \vdash \langle \mu, \{\}^{\sigma_s}, \Sigma \rangle \Downarrow_{IF} \langle \mu', r_o, \Sigma'', \sigma_{pc} \rangle}$$

CONDITIONAL

$$\frac{\begin{array}{c}r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \hat{\mu}, \hat{v}, \hat{\Sigma}, \hat{\sigma} \rangle \\ \hat{v} \notin V_F \Rightarrow i = 0 \qquad \hat{v} \in V_F \Rightarrow i = 1 \\ r, \sigma_{pc} \sqcup \hat{\sigma} \vdash \langle \hat{\mu}, e_i, \hat{\Sigma} \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle\end{array}}{r, \sigma_{pc} \vdash \langle \mu, e\ ?\ (e_0) : (e_1), \Sigma \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle}$$

SEQUENCE

$$\frac{r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, v_0, \Sigma_0, \sigma_0 \rangle \qquad r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0 \rangle \Downarrow_{IF} \langle \mu_1, v_1, \Sigma_1, \sigma_1 \rangle}{r, \sigma_{pc} \vdash \langle \mu, e_0, e_1, \Sigma \rangle \Downarrow_{IF} \langle \mu_1, v_1, \Sigma_1, \sigma_1 \rangle}$$

Figure 4.3: Monitored Core JavaScript Semantics - Imperative Fragment

its subexpressions. This rule also removes both the *value level* and the *existence level* of the property m_1 (obtained from the evaluation of e_1) in the object pointed to by r_0 (obtained from the evaluation of e_0).

- [OBJECT LITERAL] The reading effect of an object literal is simply the level of the program counter. The allocation of the new object must be paired-up with an extension of the current labelling in order to record both its *object level* and its *structure security level*.

$$\textsc{Function Call}\quad \frac{\begin{array}{c}\forall_{i=0,1}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i\rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i\rangle \\ \langle \hat{r}, \hat{\mu}, \hat{e}, \hat{\Sigma}\rangle = \mathsf{NewScope}_{lab}(\mu_2, v_0, v_1, \#glob, \Sigma_2, \sigma_0, \sigma_0 \sqcup \sigma_1) \\ \hat{r}, \sigma_0 \vdash \langle \hat{\mu}, \hat{e}, \hat{\Sigma}\rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma\rangle\end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, e_0(e_1), \Sigma_0\rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma\rangle}$$

$$\textsc{Method Call}\quad \frac{\begin{array}{c}\forall_{i=0,1,2}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i\rangle \Downarrow_{IF} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i\rangle \qquad \langle r_m, \sigma_m\rangle = \mathsf{Proto}(\mu_3, v_0, v_1, \Sigma_3) \\ r_f = \mu_3(r_m \cdot v_1) \qquad \sigma'_{pc} = \sigma_0 \sqcup \sigma_1 \sqcup \Sigma_3.\mathsf{val}(r_m \cdot v_1) \sqcup \sigma_m \\ \langle \hat{r}, \hat{\mu}, \hat{e}, \hat{\Sigma}\rangle = \mathsf{NewScope}_{lab}(\mu_3, r_f, v_2, v_0, \Sigma_3, \sigma'_{pc}, \sigma'_{pc} \sqcup \sigma_2) \\ \hat{r}, \sigma'_{pc} \vdash \langle \hat{\mu}, \hat{e}, \hat{\Sigma}\rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma\rangle\end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, e_0[e_1](e_2), \Sigma_0\rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma\rangle}$$

$$\textsc{Function Literal}\quad \frac{\begin{array}{c}r_f = \mathsf{fresh}(\sigma_{pc}) \qquad \mu' = \mu\,[r_f \mapsto [\texttt{"@fscope"} \mapsto r, \texttt{"@code"} \mapsto \lambda x.\,\{\mathsf{var}\ y_1, \cdots, y_n;\ e\}]] \\ \Sigma' = \mathsf{extend}(\Sigma, r_f, \sigma_{pc}, , \sigma_{pc}) \\ \Sigma'' = \mathsf{updt}(\Sigma', (r_f, \texttt{"@fscope"}), (\sigma_{pc}, \sigma_{pc})) \qquad \Sigma''' = \mathsf{updt}(\Sigma'', (r_f, \texttt{"@code"}), (\sigma_{pc}, \sigma_{pc}))\end{array}}{r, \sigma_{pc} \vdash \langle \mu, \mathsf{function}(x)\{\mathsf{var}\ y_1, \cdots, y_n;\ e\}, \Sigma\rangle \Downarrow_{IF} \langle \mu', r_f, \Sigma''', \sigma_{pc}\rangle}$$

Figure 4.4: Monitored Core JavaScript Semantics - Functional Fragment

Furthermore, it must also record the *value level* and the *existence level* of the property `"_prot_"` of the newly allocated object, which are both set to the current level of the program counter.

- [Conditional Expression] The reading effect of a conditional expression is the reading effect of the branch that is evaluated. During the evaluation of this branch, the level of the program counter is upgraded to the reading effect of the guard of the conditional. Hence, the reading effect of whole conditional expression is always higher than or equal to the reading effect of its guard.

- [Sequence] The reading effect of a sequence expression e_0, e_1 is the reading effect of its second subexpression.

- [Function Call] The reading effect of a function call $e_0(e_1)$ is the reading effect of the body of the function that is evaluated. The allocation of the new scope object must be paired-up with an extension of the current labelling in order for it to additionally cover the properties of the newly allocated scope object. This extension is discussed in detail in Subsection 4.1.4. During the evaluation of the body of the function, the level of the program counter is set to the reading effect of e_0.

- [Method Call] The reading effect of a method call $e_0[e_1](e_2)$ is the reading effect of the body of the method that is evaluated. Like in the case of the function call, the allocation of the new scope object must be paired-up with an extension of the current labelling in order for it to additionally cover the properties of the newly allocated scope object. During the evaluation of the body of the method, the level of the program counter is set to the *lub* between: **(1)** the reading effects of e_0 and e_1, **(2)** the level of the prototype-chain inspection procedure, **(3)** the level of the context in which the function literal corresponding to the method was evaluated, and **(4)** the *value level* of the property m_1 (obtained from the

Program:	h = 0	h = 1	
	Both Approaches	*Naive Approach*	*No-Sensitive-Upgrade*
l0 = true;	$\Sigma.\mathsf{val}(r \cdot \texttt{"l0"}) := L$	$\Sigma.\mathsf{val}(r \cdot \texttt{"l0"}) := L$	$\Sigma.\mathsf{val}(r \cdot \texttt{"l0"}) := L$
l1 = true;	$\Sigma.\mathsf{val}(r \cdot \texttt{"l1"}) := L$	$\Sigma.\mathsf{val}(r \cdot \texttt{"l1"}) := L$	$\Sigma.\mathsf{val}(r \cdot \texttt{"l1"}) := L$
h ?	branch not taken	branch taken	branch taken
(l0 = false);	—	$\Sigma.\mathsf{val}(r \cdot \texttt{"l0"}) := H$	*stuck*
l0 ?	branch taken	branch not taken	—
(l1 = false);	$\Sigma.\mathsf{val}(r \cdot \texttt{"l1"}) := L$	—	—
Final Low Memory:	l1 = false	l1 = true	—

Table 4.1: Naive Approach vs No-sensitive-upgrade

evaluation of e_1) in the object that defines a binding for m_1 in the prototype-chain of the object pointed to by r_0 (obtained from the evaluation of e_0).

- [Function Literal] The reading effect of a function literal is simply the level of the program counter. The allocation of the new function object must be paired-up with an extension of the current labelling in order for it to additionally cover the properties of the newly allocated function object: `"@fscope"` and `"@code"`. The *value level* and the *existence level* of both of these properties are set to the current level of the program counter.

4.1.1 Controlling Implicit Flows and the No-Sensitive-Upgrade Discipline

The *no-sensitive-upgrade* discipline of Zdancewic [Zdancewic 2002, Austin 2009] establishes that visible resources cannot be upgraded in invisible contexts, since such upgrades would cause the visible domain of a program to change depending on secret values. Therefore, the flow-sensitive monitors that implement the no-sensitive-upgrade discipline abort executions that encode illegal implicit flows. Intuitively, one could consider a *naive* strategy that would simply raise the security level of visible resources updated in *high* contexts to the level of the context itself. However, this strategy does not work since it would partially leak the contents of the resources on which the control flow depends.

Consider, for instance, the example given in Table 4.1 and adapted from [Austin 2010]. This table shows four monitored executions of a program (represented on the left) in two distinct memories that initially map a *high* variable `h` to 0 and 1, respectively. Specifically, one can see how the dynamic labelling Σ evolves during the execution of the program applying both the naive strategy and the no-sensitive-upgrade strategy. In the example, r is assumed to be the reference of the current scope object. While both monitors coincide on the executions starting from the memory that initially maps `h` to 0, they differ on the executions starting from the memory that initially maps `h` to 1. The monitor following the *naive* approach raises the level of `l0` to H (thus allowing the execution to go through), whereas the monitor following the *no-sensitive-upgrade* strategy blocks the execution when the program tries to update the value of `l0` in a high context. It should be noted that the execution of this program by the monitor following the *naive* strategy generates two memories that are **not** low-equal even though the initial memories are low-equal.

In Core JavaScript, there are seven types of implicit illegal flows that cause the proposed monitor to abort the execution, and they are illustrated in Table 4.2. To see why the information flows encoded in the programs given in Table 4.2 should be prevented, consider their execution

Type I	Type II	Type III	Type IV
laux = true, l = true, h ? (laux = false), laux ? (l = false)	o = { }^{L}, o.p = true, l = true, h ? (o.p = false), o.p ? (l = false)	o = { }^{L}, o.p = true, l = true, h ? (delete o.p), "p" in o ? (l = false)	oh = { }^{H}, ol = { }^{L}, l = true, ol.p = false, h ? (oh = ol), oh.p = true, !ol.p ? (l = false)

Type V	Type VI	Type VII
l = true, o = { }^{H}, o.q = false, proph = "p", h ? (proph = "q"), o[proph] = true, !o.q ? (l = false)	oh = { }^{H}, ol = { }^{L}, oh.p = true, ol.p = true, l = true, h ? oh = ol, delete oh.["p"], "p" in ol ? (l = false)	o = { }^{H}, proph = "q", o.p = true, o[proph] = true, l = true, h ? proph = "p", delete o[proph], "p" in o ? (l = false)

Table 4.2: Naive Approach vs No-sensitive-upgrade

by a monitor following the *naive* approach in two memories that initially map a *high* variable h to 0 and 1, respectively. The execution of all six programs in a memory that originally maps h to 0 terminates with a memory that maps the low variable l to false (without raising its security level to H). Alternatively, their execution in a memory that originally maps h to 1 terminates with a memory that maps the low variable l to true (without raising its security level to H). Since the two initial memories are low-equal, one can see that the execution of these programs by a monitor following the naive strategy reveals information about the secret contents of the initial memory (specifically, the content of the *high* variable h). Below, we list and briefly comment on each of the types of illegal implicit flow:

- **Visible Variable Assignment in an Invisible Context (Type I):** the monitor blocks assignments to variables holding visible values in *high* contexts. Therefore, in the example, the monitor blocks the assignment of false to laux inside the first conditional.
- **Visible Property Assignment in an Invisible Context (Type II):** the monitor blocks assignments to properties holding visible values within invisible contexts. Therefore, in the example, the monitor blocks the assignment of false to o.p inside the first conditional.
- **Visible Property Deletion in an Invisible Context (Type III):** the monitor blocks deletions of visible properties in invisible contexts. Therefore, in the example, the monitor blocks the deletion of the property "p" of the object bound to o inside the first conditional.
- **Visible Property Assignment via Invisible Reference (Type IV):** the monitor blocks assignments to visible properties when the reference pointing to the object that binds the property was computed using secret information. For instance, in the example,

while the *low* variable `ol` can only hold *low* references, the *high* variable `oh` can hold both *low* and *high* references. Therefore, the assignment `oh = ol` is allowed to go through. However, when `oh` is set to point to the same reference as `ol`, the assignment `oh.p = true` is blocked, since it tries to update the value of a *low* property via a *high* reference.

- **Visible Property Assignment via an Invisible Property Name (Type V):** the monitor blocks assignments to visible properties when the corresponding property name was computed using secret information. For instance, in the example, the variable `proph` can hold both *low* and *high* property names. Therefore, the assignment `proph = "q"` is allowed to go through, even though it is performed inside a *high* conditional. However, after this assignment, the assignment `o[proph] = true` is blocked since it tries to update the value of a *low* property via a *high* property name.

- **Visible Property Deletion via an Invisible Reference (Type VI):** the monitor blocks the deletion of visible properties when the reference pointing to the object that binds the property was computed using secret information. For instance, in the example, the *high* variable `oh` can hold both *low* references and *high* references. Therefore, the assignment `oh = ol` is allowed to go through. However, when `oh` is set to point to the same reference as `ol`, the execution of `delete oh[`p`]` is blocked since it constitutes a *low* property deletion via a *high* reference.

- **Visible Property Deletion via an Invisible Property Name (Type VII):** the monitor blocks the deletion of a visible property when the corresponding property name was computed using secret information. For instance, in the example, the *high* variable `proph` can hold both *low* property names and *high* property names. Therefore, the assignment `proph = "p"` is allowed to go through inside the body of the *high* conditional. However, when `proph` is set to `"p"`, the execution of `delete o[proph]` is blocked since it constitutes a *low* property deletion via a *high* property name.

4.1.2 The Structure Security Level

Since objects in Core JavaScript are initially created without any properties, the structure security level of an object defines an upper bound on the existence levels of the properties that can be added to that object. In this sense, *the structure security level of an object can be understood as the security level associated with its domain.* If an object has a *low* structure security level, all its properties have *low* existence levels, which means that the entire domain of the object is visible. If an object has a *high* structure security level, only its properties with *low* existence levels are visible. Again, we emphasise that the structure security level is not a key element for the characterisation of the attacker model inherent to JavaScript, but rather a device of the enforcement mechanism. The need for the structure security level arises from the fact that existence levels are not established *a priori*.

Since the structure security level is used to control the **implicit information flows** that can be encoded by modifying the domain of an object, it cannot be upgraded. In fact, such upgrades would violate the no-sensitive-upgrade discipline, which forbids upgrades based on implicit flows. Hence, if an object o has a *low* structure security level, one can only change its structure (either by adding properties to o or removing properties from o) in *low* contexts.This fact is illustrated in Table 4.3, which shows four monitored executions of a program in two distinct memories that initially map a *high* variable `h` to 0 and 1 respectively. While both monitors coincide on the executions starting from the memory that initially maps `h` to 0, they differ on the executions starting from the memory that initially maps `h` to 1. The monitor following the *naive* approach raises the structure security level of the object bound to `o` to H (thus allowing the execution to go

Program:	h = 0	h = 1	
	Both Approaches	*Naive Approach*	*No-Sensitive-Upgrade*
l = true;	$\Sigma.\mathsf{val}(r \cdot \texttt{"l"}) := L$	$\Sigma.\mathsf{val}(r \cdot \texttt{"l"}) := L$	$\Sigma.\mathsf{val}(r \cdot \texttt{"l"}) := L$
o = { }^{L};	$\Sigma.\mathsf{val}(r \cdot \texttt{"o"}) := L/$ $\Sigma.\mathsf{struct}(r_o) := L$	$\Sigma.\mathsf{val}(r \cdot \texttt{"o"}) := L/$ $\Sigma.\mathsf{struct}(r_o) := L$	$\Sigma.\mathsf{val}(r \cdot \texttt{"o"}) := L/$ $\Sigma.\mathsf{struct}(r_o) := L$
h ?	branch not taken	branch taken	branch taken
(o.p = true);	—	$\Sigma.\mathsf{val}(r_o \cdot \texttt{"p"}) := H/$ $\Sigma.\mathsf{exist}(r_o \cdot \texttt{"p"}) := H/$ $\Sigma.\mathsf{struct}(r_o) := L$	*stuck*
!("p" in o) ?	branch taken	branch not taken	—
(l = false);	$\Sigma.\mathsf{val}(r \cdot \texttt{"l"}) := L$	—	—
Final Low Memory:	l = false	l = true	—

Table 4.3: Preventing Security Leaks via the Domain of an Object

through), whereas the monitor following the *no-sensitive-upgrade* strategy blocks the execution when the program tries to create a property in an object with a *low* structure security level inside a *high* context. We assume in this example that the created object is stored in reference r_o and that r is the reference of the current scope object. Observe that the execution of this program by the monitor following the *naive* strategy generates two memories that are **not** low-equal even though the initial memories are low-equal.

4.1.3 Preventing Security Leaks via Prototype Mutations

Let us suppose that a program looks up the value of a property p in an object o, and that $p \notin dom(o)$. Then, since this property look-up leaks information about the domain of o (one gets to know that p does not belong to the domain of o), the security level associated with the property look-up expression must be equal to or higher than the structure security level of o. Furthermore, it must also be higher than or equal to the level of o `"_prot_"` property, since the value of this property determines what the object that the prototype-chain look-up procedure will inspect next is. In fact, the security monitor has to take into account the structure security level as well as the level of the `"_prot_"` property of every object traversed during the prototype-chain inspection procedure until it finds the object that defines a binding for the property being looked-up. For example, given a memory:

$$\mu = \begin{bmatrix} \#o_0 \mapsto [\texttt{"xpto"} \mapsto 1, \texttt{"\_prot\_"} \mapsto \mathsf{null}], \\ \#o_1 \mapsto [\texttt{"\_prot\_"} \mapsto \#o_0], \\ \#glob \mapsto [\texttt{"o1"} \mapsto \#o_1] \end{bmatrix} \tag{4.1}$$

and a labelling Σ, such that either $\Sigma.\mathsf{struct}(\#o_0) = H$ or $\Sigma.\mathsf{val}(\#o_0 \cdot \texttt{"\_prot\_"}) = H$, the reading effect of the expression `o1.xpto` must be H, because it leaks information both about the domain of the object bound to `o1` and about its prototype. In the next definition, we redefine the prototype-chain look-up procedure in order to additionally compute the security level associated with the prototype-chain inspection procedure.

Definition 4.1 (Proto). *The semantic function* Proto : Mem × Ref × Str × *Lab* → Ref × $\mathcal{L}$ *is*

recursively defined as follows:

$$\mathsf{Proto}(\mu, r, p, \Sigma) = \begin{cases} (\mathsf{null}, \bot) & \text{if } r = \mathsf{null} \\ (r, \Sigma.\mathsf{exist}(r \cdot m)) & \text{if } p \in dom(\mu(r)) \\ (r', \Sigma.\mathsf{val}(r \cdot \texttt{"\_prot\_"}) \sqcup \Sigma.\mathsf{struct}(r) \sqcup \sigma) & \text{if } r \neq \mathsf{null} \ \wedge \ p \notin dom(\mu(r)) \end{cases}$$

where $(r', \sigma) = \mathsf{Proto}(\mu, \mu(r \cdot \texttt{"\_prot\_"}), p, \Sigma)$.

4.1.4 Tracking the Level of the Program Counter

An information flow monitor must keep track of *the level of the program counter* in order to prevent illegal implicit flows. In the particular case of Core JavaScript, the level of the program counter must always be higher than or equal to the security levels of the resources that were used to decide the following:

- which branch to take in a conditional expression whose code is still being executed,
- which function/method to execute in a function/method call expression whose code is still being executed.

In order to cater for the first point, when a branch of a conditional expression is evaluated, the level of the program counter is upgraded to the reading effect of its guard. On the other hand, when calling a function/method, the level of the program counter must be upgraded to the *lub* between the reading effects of the expressions that were used to decide which function/method was to be called. In order to illustrate these two points, consider the execution of the following program in a memory that originally maps the *low* global variables `l1` and `l2` to 0.

$$\begin{array}{l} \texttt{f1} = \mathsf{function}(\texttt{x})\{\texttt{l1} = 1\}, \\ \texttt{f2} = \mathsf{function}(\texttt{x})\{\texttt{l2} = 1\}, \\ \texttt{h} \ ? \ (\texttt{f} = \texttt{f1}) \ : \ (\texttt{f} = \texttt{f2}), \\ \texttt{f}() \end{array} \tag{4.2}$$

Assuming that the security level of `h` is originally set to *high*, after the execution of this program both variables `l1` and `l2` depend on the initial value of variable `h` (independently of which function gets executed). However, since the monitor is purely dynamic, it cannot upgrade the levels of the resources that are not updated at runtime. Hence, the execution of this program must always be blocked by the monitor. This is precisely what happens, since the level of the program counter is upgraded to the level of `f` during the execution of the function bound to `f`. The level of `f` must be *high*, otherwise the assignments of the then-branch and of the else-branch of the conditional expression are blocked.

Labelled New Scope Allocation The semantic function $\mathsf{NewScope}_{lab} : \texttt{Mem} \times \texttt{Ref} \times \texttt{Val} \times \texttt{Ref} \times \texttt{Lab} \times \mathcal{L} \times \mathcal{L} \rightarrow \texttt{Mem} \times \texttt{Val} \times \texttt{Ref} \times \texttt{Lab}$ is used by the monitored semantics to allocate a new **labelled** scope object. As its unmonitored counterpart NewScope, $\mathsf{NewScope}_{lab}$ allocates a new scope object. In addition, it updates the current labelling with the security labels associated with the newly allocated scope object and its properties. Given the statement $\langle r', \mu', e, \Sigma' \rangle = \mathsf{NewScope}_{lab}(\mu, r_f, v_{arg}, r_{this}, \Sigma, \sigma_{pc}, \sigma_{arg})$, we can read that that Σ' is the labelling resulting from the extension of Σ to the newly allocated scope object. Concretely, the value level of the property matching the name of the formal argument of the function to execute is set to σ_{arg}. All of the other value levels and existence levels are set to σ_{pc} — an upper bound on the level of the resources that were used to decide which function was to be executed. The remaining elements keep their original meaning.

Definition 4.2 ($\mathsf{NewScope}_{lab}$)**.** *For any two memories μ and μ', three references r_f, r_{this}, and r', a value v_{arg}, an expression e, security levels σ_{arg} and σ_{pc}, and labellings Σ and Σ', $\langle r', \mu', e, \Sigma' \rangle = \mathsf{NewScope}_{lab}(\mu, r_f, v_{arg}, r_{this}, \Sigma, \sigma_{pc}, \sigma_{arg})$ holds if and only if:*

- $\lambda x.\, \{\mathsf{var}\ y_1, \cdots, y_n;\ e\} = \mu(r_f \cdot \texttt{"@code"})$,
- $r = \mu(r_f \cdot \texttt{"@fscope"})$,
- $r' = \mathsf{fresh}(\sigma_{pc})$,
- $\mu' = \mu\,[r' \mapsto [\texttt{"@scope"} \mapsto r, m_x \mapsto v_{arg}, \texttt{"@this"} \mapsto r_{this}, m_{y_1} \mapsto \mathsf{undefined}, \cdots, m_{y_n} \mapsto \mathsf{undefined}]]$, *where:* $m_x = \mathsf{string}(x)$, $m_{y_1} = \mathsf{string}(y_1)$, *...,* *and* $m_{y_n} = \mathsf{string}(y_n)$,
- $\Sigma'.\mathsf{obj} = \Sigma.\mathsf{obj}\,[r' \mapsto \sigma_{pc}]$,
- $\Sigma'.\mathsf{exist} = \Sigma.\mathsf{exist}\,[r' \mapsto [\texttt{"@fscope"} \mapsto \sigma_{pc}, m_x \mapsto \sigma_x, \texttt{"@this"} \mapsto \sigma_{pc}, m_{y_1} \mapsto \sigma_{pc}, \cdots, m_{y_n} \mapsto \sigma_{pc}]]$,
- $\Sigma'.\mathsf{val} = \Sigma.\mathsf{val}\,[r' \mapsto [\texttt{"@fscope"} \mapsto \sigma_{pc}, m_x \mapsto \sigma_x, \texttt{"@this"} \mapsto \sigma_{pc}, m_{y_1} \mapsto \sigma_{pc}, \cdots, m_{y_n} \mapsto \sigma_{pc}]]$,
- $\Sigma'.\mathsf{struct} = \Sigma.\mathsf{struct}\,[r' \mapsto \sigma_{pc}]$

for some variables $x, y_1, \cdots, y_n$ such that $m_x = \mathsf{string}(x)$ and $m_{y_i} = \mathsf{string}(y_i)$, for $i \in \{1, \ldots, n\}$.

4.1.5 Monitor Noninterference

Classically [Volpano 1996], one of the first steps towards proving a noninterference result is to establish a *confinement result*. In the present case, Theorem 4.1 establishes that the monitored execution of a Core JavaScript expression in a *high* context does **not** update or create *low* memory. Therefore, when executing a Core JavaScript program using the monitor in a *high* context, the low-projections of the initial and final memories coincide.

Theorem 4.1 (Confinement)**.** *Given an expression e, a memory μ, a labelling Σ, a level σ_{pc}, and a reference r such that: $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle$ for some memory μ', value v, labelling Σ' and security level σ; then for every security level $\sigma' \in \mathcal{L}$ such that $\sigma_{pc} \not\sqsubseteq \sigma'$: $\mu, \Sigma \sim_{\sigma'} \mu', \Sigma'$.*

Informally, we say that a security monitor is *noninterferent* if successfully terminating monitored executions always preserve the low-equality relation. More precisely, an information flow monitor is noninterferent *if and only if*, for any expression e, whenever an attacker cannot distinguish two labelled memories before executing e, then the attacker is also unable to distinguish the memories that result from the monitored execution of e. Hence, an attacker cannot use the monitored execution of a program as a means to obtain information about the confidential contents of a memory. Theorem 4.2 states that the monitored successfully-terminating execution of a program on two low-equal memories always yields two low-equal memories.

Theorem 4.2 (Noninterferent Monitor)**.** *For any expression e, memories μ and μ', respectively labelled by Σ and Σ', reference r, and security levels σ_{pc} and σ, such that:*

- $\mu, \Sigma \sim_\sigma \mu', \Sigma'$,
- $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$,
- $r, \sigma_{pc} \vdash \langle \mu', e, \Sigma' \rangle \Downarrow_{IF} \langle \mu'_f, v'_f, \Sigma'_f, \sigma'_f \rangle$;

Then: $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ and $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$.

The second claim of the theorem states that, whenever one of the executions produces a visible value, the other also produces a visible value and the two values coincide.

The **proofs** of the results presented in this section are given in **Appendix A.1**.

Labelled Object	Instrumented Object
$o = [p \mapsto v_0, q \mapsto v_1]$ $\Sigma.\mathsf{val}(r_o) = [p \mapsto \sigma_p, q \mapsto \sigma_q]$ $\Sigma.\mathsf{exist}(r_o) = [p \mapsto \sigma'_p, q \mapsto \sigma'_q]$, $\Sigma.\mathsf{struct}(r_o) = \sigma_s$	$\hat{o} = \left[\begin{array}{l} p \mapsto v_0, \$p \mapsto \sigma_p, \$\bar{p} \mapsto \sigma'_p, \\ q \mapsto v_1, \$q \mapsto \sigma_q, \$\bar{q} \mapsto \sigma'_q, \\ \texttt{"\$struct"} \mapsto \sigma_s \end{array}\right]$

Table 4.4: Labelled Object vs. Instrumented Object

4.2 Monitor-Inlining

This section presents an information flow monitor-inlining compiler for Core JavaScript. The proposed compiler instruments programs in order to simulate their execution in the monitored semantics presented in Section 4.1. This instrumentation is based on a technique that consists of pairing up each variable with a *shadow* variable [Magazinius 2012, Chudnov 2010] that holds its corresponding security level, and each property with two *shadow* properties that hold its corresponding value level and existence level. As the compiled program has to handle security levels, we include these shadow variables and properties in the set of program values, effectively adding them to the syntax of the language as such. We also add two new binary operators corresponding to the order relation ($\sqsubseteq$) and the least upper bound ($\sqcup$) between security levels.

In the design of the compiler, we assume the existence of two given sets of *internal* variables and property names, denoted by $\mathtt{S}_{comp}$, that do not overlap with those available for the programmer. In particular, the compilation of every expression requires additional variables intended to bookkeep the value to which it evaluates and its reading effect. These variables are later used in the compilation of the expressions that include this expression. Hence, we assume the set of compiler variables to include two indexed sets of variables $\{\$v_i\}_{i \in \mathbf{N}}$ and $\{\$l_i\}_{i \in \mathbf{N}}$ used to store the levels and the values of intermediate expressions, respectively. For clarity, all identifiers that are reserved for the compiler are prefixed with the dollar sign.

For each variable x, the compiler adds a new *shadow* variable, $\$x$, that holds its corresponding security level. Analogously, for each property p, the compiler adds two new properties, $\$p$ and $\$\bar{p}$, that hold its corresponding *value level* and *existence level*. In contrast to variables, whose names are available at compile time, property names can be dynamically computed. Therefore, we assume the existence of two runtime functions, bound to the variables `$shadowV` and `$shadowE`, that, given a property name, output the name of the shadow properties that hold its value level and existence level, respectively. Concretely, given a property p, the expression `$shadowV`(p) evaluates to $\$p$ and the expression `$shadowE`(p) evaluates to $\$\bar{p}$.

Apart from adding to every object o two additional shadow properties $\$p$ and $\$\bar{p}$ for every property p in its domain, the inlined monitoring code also adds to o a special property `"$struct"` that stores its structure security level. Table 4.4 represents a labelled object o (pointed to by a reference r_o) on the left and its instrumented counterpart on the right.

4.2.1 Malicious Code

Given an expression e to compile, the compiler guarantees that e does not use variable and property names in $\mathtt{S}_{comp}$ by:

1. statically verifying that the names of the variables in e do not overlap with $\mathtt{S}_{comp}$,
2. dynamically verifying that e does not look-up, create, update, or delete properties whose names belong to $\mathtt{S}_{comp}$.

In order to perform the dynamic check, the compiler makes use of a runtime function bound to the variable `$legal` that returns `true` when its argument does not belong to $\mathtt{S}_{comp}$.

By making sure that compiler identifiers do not overlap with the identifiers of the programs to compile, we guarantee the soundness of the proposed transformation even when it receives as input *malicious programs*. Malicious programs attempt to bypass the inlined runtime enforcement mechanism by rewriting some of its internal variables/properties. For instance, considered the following program that is to be executed in a memory that originally maps x_h to a secret value and x_l to a public value:

$$\$x_h = L,\ x_l = x_h \tag{4.3}$$

This program tries to tamper with the internal state of the runtime enforcement mechanism in order to be allowed to leak confidential information. Concretely, it tries to transfer the content of x_h to x_l without raising the level of x_l. To this end, it first sets the level of x_h (stored in variable $\$x_h$) to L (*low*). However, the compiler statically detects that the program makes use of an identifier reserved for the runtime enforcement mechanism and the compilation fails.

4.2.2 Formal Specification

The inlining compiler is defined as a function $\mathcal{C}$, given in Figures 4.5 and 4.6. It expects as input an expression e and produces a pair $\langle \hat{e} \mid i \rangle$, where $\hat{e}$ is the expression that simulates the execution of e in the monitored semantics and i an index such that, after the execution of $\hat{e}$, $\$v_i$ stores the value to which e evaluates and $\$l_i$ its corresponding reading effect. Besides the runtime functions bound to `$shadowV`, `$shadowE`, and `$legal`, the compiler makes use of:

- a runtime function bound to `$check` that diverges when its argument is different from `true`;
- a runtime function bound to `$inspect` that expects as input an object and a property and outputs the level associated with the corresponding prototype-chain inspection procedure;
- an additional binary operator hasOwnProp that checks whether the object given as its left operand defines the property given as its right one.

In JavaScript, the operator hasOwnProp does not exist; instead, there exists a method *hasOwnProperty*, accessible to every object *via* its corresponding prototype chain, that checks whether the object on which it is invoked defines the property whose name it receives as input. We chose not to model this feature of the language exactly as it is in the specification in order to keep the model as simple as possible. Doing it otherwise would imply cluttering the already complex semantics of Core JavaScript by having an alternative case for the Rule [Method Call], which would model the semantics of *hasOwnProperty*.

During the evaluation of the instrumented code, the level of the execution context (σ_{pc}) is assumed to be stored in the variable `$pc`. To this end, function literals are instrumented in order to receive as input the level of the argument and the level of the context in which they are invoked. Function/method calls are instrumented accordingly. Furthermore, the instrumented code of a function/method call must have access to both the return value of the original function/method and the level that is to be associated with that value. Therefore, every function literal returns an object that defines two properties: **(1)** a property `"val"` that stores the return value of the original function and **(2)** a property `"lev"` that stores the level to be associated with that value.

Each compiler rule precisely mimics the corresponding monitor rule. As done in the presentation of the monitor, constraints are depicted in red and labelling updates are depicted in orange. The compiled code must bookkeep the level and value of indexed expressions. To this end, given an expression e with index i, the compilation of e assigns the value to which it evaluates to a

VALUE

$$\frac{\hat{e} = \begin{cases} \$l_i = \$\mathtt{pc}, \\ \$v_i = v \end{cases}}{\mathcal{C}\langle v^i \rangle = \langle \hat{e} \mid i \rangle}$$

VARIABLE

$$\frac{\mathsf{string}(x) \notin \mathsf{S}_{comp} \qquad \hat{e} = \begin{cases} \$l_i = \$\mathtt{pc} \sqcup \$x, \\ \$v_i = x \end{cases}}{\mathcal{C}\langle x^i \rangle = \langle \hat{e} \mid i \rangle}$$

THIS

$$\frac{\hat{e} = \begin{cases} \$l_i = \$\mathtt{pc}, \\ \$v_i = \mathsf{this} \end{cases}}{\mathcal{C}\langle \mathsf{this}^i \rangle = \langle \hat{e} \mid i \rangle}$$

BINARY OPERATION

$$\frac{\langle \hat{e}_0 \mid j \rangle = \mathcal{C}\langle e_0 \rangle \qquad \langle \hat{e}_1 \mid k \rangle = \mathcal{C}\langle e_1 \rangle \qquad \hat{e} = \begin{cases} \hat{e}_0, \\ \hat{e}_1, \\ \$l_i = \$l_j \sqcup \$l_k, \\ \$v_i = \$v_j \ \mathsf{op} \ \$v_k \end{cases}}{\mathcal{C}\langle e_0 \ \mathsf{op}^i \ e_1 \rangle = \langle \hat{e} \mid i \rangle}$$

VARIABLE ASSIGNMENT

$$\frac{\mathsf{string}(x) \notin \mathsf{S}_{comp} \qquad \langle e' \mid i \rangle = \mathcal{C}\langle e \rangle \qquad \hat{e} = \begin{cases} e' \\ \mathtt{\$check(\$pc \sqsubseteq \$x)}, \\ \$x = \$l_i, \\ x = \$v_i \end{cases}}{\mathcal{C}\langle x = e \rangle = \langle e', \hat{e} \mid i \rangle}$$

PROPERTY LOOK-UP

$$\frac{\langle \hat{e}_0 \mid j \rangle = \mathcal{C}\langle e_0 \rangle \qquad \langle \hat{e}_1 \mid k \rangle = \mathcal{C}\langle e_1 \rangle \qquad \hat{e} = \begin{cases} \hat{e}_0, \\ \hat{e}_1, \\ \$l_i = \$l_j \sqcup \$l_k \sqcup \mathtt{\$inspect}(\$v_j, \$v_k), \\ (\$v_k \ \mathsf{in} \ \$v_j) \ ? \\ \quad (\$l_i = \$l_i \sqcup \$v_j[\mathtt{\$shadowV}(\$v_k)]), \\ \mathtt{\$check(\$legal(\$}v_k\mathtt{))}, \\ \$v_i = \$v_j[\$v_k] \end{cases}}{\mathcal{C}\langle e_0[e_1]^i \rangle = \langle \hat{e} \mid i \rangle}$$

MEMBERSHIP TESTING

$$\frac{\langle \hat{e}_0 \mid j \rangle = \mathcal{C}\langle e_0 \rangle \qquad \langle \hat{e}_1 \mid k \rangle = \mathcal{C}\langle e_1 \rangle \qquad \hat{e} = \begin{cases} \hat{e}_0, \ \hat{e}_1, \\ \$l_i = \$l_j \sqcup \$l_k \sqcup \mathtt{\$inspect}(\$v_k, \$v_j), \\ (\$v_j \ \mathsf{in} \ \$v_k) \ ? \\ \quad (\$l_i = \$l_i \sqcup \$v_k[\mathtt{\$shadowE}(\$v_j)]), \\ \mathtt{\$check(\$legal(\$}v_j\mathtt{))}, \\ \$v_i = \$v_j \ \mathsf{in} \ \$v_k \end{cases}}{\mathcal{C}\langle e_0 \ \mathsf{in}^i \ e_1 \rangle = \langle \hat{e} \mid i \rangle}$$

PROPERTY ASSIGNMENT

$$\frac{\langle \hat{e}_0 \mid i \rangle = \mathcal{C}\langle e_0 \rangle \qquad \langle \hat{e}_1 \mid j \rangle = \mathcal{C}\langle e_1 \rangle \qquad \langle \hat{e}_2 \mid k \rangle = \mathcal{C}\langle e_2 \rangle \qquad \hat{e} = \begin{cases} \hat{e}_0, \ \hat{e}_1, \ \hat{e}_2, \ \mathtt{\$check(\$legal(\$}v_j\mathtt{))}, \\ (\$v_i \ \mathsf{hasOwnProp} \ \$v_j) \ ? \\ \quad (\mathtt{\$check(\$}l_i \sqcup \$l_j \sqsubseteq \$v_i\mathtt{[\$shadowV(\$}v_j\mathtt{)])}) \\ \quad : \ (\mathtt{\$check(\$}l_i \sqcup \$l_j \sqsubseteq \$v_i\mathtt{.\$struct)}, \\ \qquad \$v_i[\mathtt{\$shadowE(\$}v_j\mathtt{)}] = \$l_i \sqcup \$l_j), \\ \$v_i[\mathtt{\$shadowV(\$}v_j\mathtt{)}] = \$l_i \sqcup \$l_j \sqcup \$l_k, \\ \$v_i[\$v_j] = \$v_k \end{cases}}{\mathcal{C}\langle e_0[e_1] = e_2 \rangle = \langle \hat{e} \mid k \rangle}$$

OBJECT LITERAL

$$\frac{\hat{e} = \begin{cases} \$v_i = \{\}, \\ \$v_i\mathtt{.\$struct} = \sigma_s, \\ \$v_i[\mathtt{\$shadowE("\_prot\_")}] = \$\mathtt{pc}, \\ \$v_i[\mathtt{\$shadowV("\_prot\_")}] = \$\mathtt{pc}, \\ \$l_i = \$\mathtt{pc}, \\ \$v_i \end{cases}}{\mathcal{C}\langle \{\}^{i,\sigma_s} \rangle = \langle \hat{e} \mid i \rangle}$$

PROPERTY DELETION

$$\frac{\langle \hat{e}_0 \mid j \rangle = \mathcal{C}\langle e_0 \rangle \qquad \langle \hat{e}_1 \mid k \rangle = \mathcal{C}\langle e_1 \rangle \qquad \hat{e} = \begin{cases} \hat{e}_0, \ \hat{e}_1, \\ \mathtt{\$check(\$}l_j \sqcup \$l_k \sqsubseteq \$v_j\mathtt{[\$shadowE(\$}v_k\mathtt{)])}, \\ \mathsf{delete} \ \$v_j[\mathtt{\$shadowE(\$}v_k\mathtt{)}], \\ \mathsf{delete} \ \$v_j[\mathtt{\$shadowV(\$}v_k\mathtt{)}], \\ \$l_i = \$\mathtt{pc}, \\ \$v_i = \mathsf{delete} \ \$v_j[\$v_k] \end{cases}}{\mathcal{C}\langle \mathsf{delete}^i \ e_0[e_1] \rangle = \langle \hat{e} \mid i \rangle}$$

CONDITIONAL

$$\frac{\langle \hat{e}_0 \mid i \rangle = \mathcal{C}\langle e_0 \rangle \qquad \langle \hat{e}_1 \mid j \rangle = \mathcal{C}\langle e_1 \rangle \qquad \langle \hat{e}_2 \mid k \rangle = \mathcal{C}\langle e_2 \rangle \qquad \hat{e} = \begin{cases} \hat{e}_0, \ \$l_s = \$\mathtt{pc}, \ \$\mathtt{pc} = \$\mathtt{pc} \sqcup \$l_i, \\ \$v_i \ ? \\ \quad (\hat{e}_1, \$v_t = \$v_j, \ \$l_t = \$l_j) \\ \quad : \ (\hat{e}_2, \$v_t = \$v_k, \ \$l_t = \$l_k), \\ \$\mathtt{pc} = \$l_s, \ \$v_t \end{cases}}{\mathcal{C}\langle e_0 \ ?^{s,t} \ (e_1) : (e_2) \rangle = \langle \hat{e} \mid t \rangle}$$

SEQUENCE

$$\frac{\langle \hat{e}_0 \mid i \rangle = \mathcal{C}\langle e_0 \rangle \qquad \langle \hat{e}_1 \mid j \rangle = \mathcal{C}\langle e_1 \rangle}{\mathcal{C}\langle e_0, e_1 \rangle = \langle \hat{e}_0, \hat{e}_1 \mid j \rangle}$$

Figure 4.5: Monitor-Inlining Compiler - Imperative Fragment

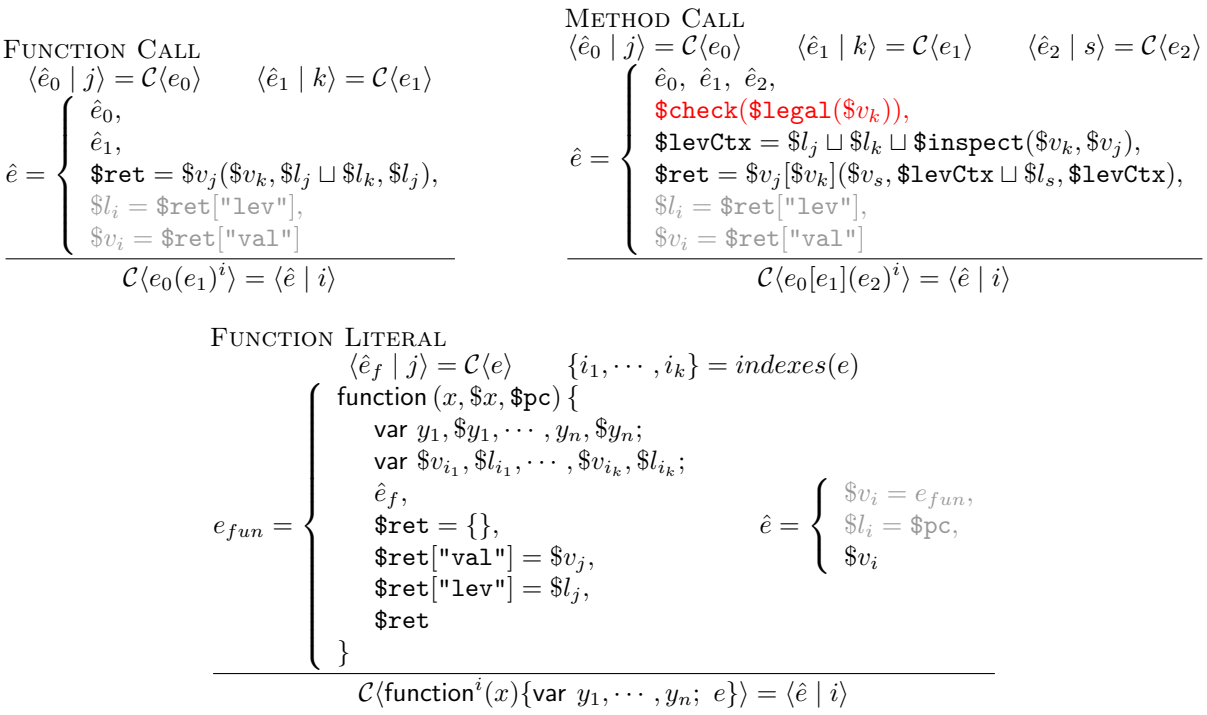


Figure 4.6: Monitor-Inlining Compiler - Functional Fragment

new variable $\$v_i$ and its reading effect to a new variable $\$l_i$. We use light grey for depicting bookkeeping code. The compilation of variable/property assignments and sequence expressions does not introduce additional variables because the corresponding value and reading effect are already available through the indexed variables introduced by the corresponding subexpressions.

4.2.3 Correctness

In Definition 4.3, we present a *similarity relation* $\mathcal{S}$ between labelled memories in the monitored semantics and instrumented memories in the original semantics. This relation requires that for every object in the labelled memory, its corresponding labelling coincides with its instrumented labelling (except for some internal properties whose levels can be automatically inferred) and that the property values of the original object coincide with those of its instrumented counterpart.

Definition 4.3 (Memory Similarity)**.** *A memory μ labelled by Σ is similar to a memory μ', written $\mu, \Sigma\ \mathcal{S}\ \mu'$, if and only if, for every reference $r \in dom(\mu)$, it holds that:*

- $\forall_{p \in dom(\mu(r))}\ \mu(r \cdot p) = \mu'(r \cdot p)$,
- $\forall_{p \in @dom(\mu(r))}\ \Sigma.\mathsf{val}(r \cdot p) = \mu'(r \cdot \$p)$, *and*
- *If $\mu(r)$ is* **not internal**: $\forall_{p \in dom(\mu(r))}\ \Sigma.\mathsf{exist}(r \cdot p) = \mu'(r \cdot \$\bar{p})$ *and* $\Sigma.\mathsf{struct}(r) = \mu'(r \cdot \mathtt{"\$struct"})$.

The Correctness Theorem states that, provided that a program and its compiled counterpart are evaluated in similar configurations, the evaluation of the original one in the monitored semantics terminates *if and only if* the evaluation of its compilation terminates in the original semantics. Additionally, if that happens, the final memories are similar and the computed values coincide. Therefore, since the monitored semantics only allows secure executions to go through, we guarantee that, when using the inlining compiler, programs are rewritten in such a way that only their secure executions are allowed to terminate.

Theorem 4.3 (Correctness)**.** *Provided that e does not use identifiers in* $\mathtt{S}_{comp}$*, for any labelled and instrumented configurations* $\langle \mu, e, \Sigma \rangle$ *and* $\langle \mu', e' \rangle$*, such that* $\mu, \Sigma \; \mathcal{S} \; \mu'$ *and* $\mathcal{C}\langle e \rangle = \langle e' \mid i \rangle$*, for some index i, and for any reference r in* $dom(\mu)$ *such that* $\mu'(r \cdot \texttt{"\$pc"}) = \bot$*, it is always the case that:*

$$\begin{array}{l} \exists \langle \mu_f, v, \Sigma_f, \sigma \rangle \\ \quad r, \bot \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \mu_f, v, \Sigma, \sigma \rangle \end{array} \quad \textbf{iff} \quad \begin{array}{l} \exists \langle \mu'_f, v' \rangle \\ \quad r \vdash \langle \mu', e' \rangle \Downarrow \langle \mu'_f, v' \rangle \end{array}$$

Moreover, if either of the two sides of the equivalence holds, then:

- $\mu_f, \Sigma_f \; \mathcal{S} \; \mu'_f$,
- $v = v' = \mu'_f(r \cdot \$v_i)$*, and*
- $\sigma = \mu'_f(r \cdot \$l_i)$.

The **proof** of Theorem 4.3 is given in **Appendix A.2**.

4.3 Related Work

Dynamic Monitors for Enforcing Secure Information Flow. Flow-sensitive monitors for enforcing noninterference can be broadly divided into two classes: those that are *purely dynamic*, such as [Zdancewic 2002] [Austin 2009], [Austin 2010], and [Austin 2012], and those commonly referred to as *hybrid monitors*, that mix runtime monitoring with static analysis, such as [Venkatakrishnan 2006], [Guernic 2007], and [Shroff 2007]. In contrast to hybrid monitors,[1] purely dynamic monitors do not rely on any kind of static analysis. Instead, the authors of [Austin 2009], [Austin 2010], and [Austin 2012] propose three alternative strategies for designing sound purely dynamic information flow monitors.

- The *no-sensitive-upgrade* strategy [Zdancewic 2002, Austin 2009], that forbids the update of public resources inside private contexts.
- The *permissive-upgrade* [Austin 2010] strategy, that allows sensitive upgrades to take place, but marks the resources upgraded in sensitive contexts and forbids the program to branch depending on the content of these resources.
- Finally, the *multiple facet* strategy surpasses the limitations of the first two by the use of multiple faceted values. The intuition behind this strategy is that values must appear differently to observers at different security levels. Therefore, the security monitor simulates multiple executions for different security levels.

It is unclear whether or not the multiple facet strategy should be considered as a purely dynamic approach, since, despite not performing any kind of static analysis, it does perform *look aside* operations [Russo 2010]. In other words, it may dynamically inspect program branches that are not executed.

Hybrid monitors must estimate, either statically or dynamically, the set of resources that are updated/created in untaken program branches. Our choice for the inlining of a purely dynamic monitor has to do with the fact that the dynamic features of JavaScript make it very difficult to compute such an approximation. Therefore, we have chosen to start with a simpler goal, which can be viewed as a first step in that direction.

[1] The literature review regarding hybrid monitors is deferred to the related work section of Chapter 5, where we provide an overview of hybrid analyses for securing information flow.

Coarse Grained Information Flow Monitors. In the past several years, *coarse-grained* information flow monitors [Russo 2008, Stefan 2011, Stefan 2014, Buiras 2014] have emerged as an alternative to fine-grained information flow monitors. The main advantage of this type of monitor with respect to fine-grained monitors is that they are easier to integrate with existing languages [Russo 2008, Stefan 2014]. In fact, in monadic languages such as Haskell, this type of monitor can even be implemented as a library [Russo 2008]. The drawback of this approach, however, is that it comes at the cost of losing precision.

The design of coarse-grained information flow monitors was inspired by information flow control mechanisms for operating systems. Concretely, coarse-grained monitors are designed in a way such that the level of the program counter represents an upper bound on the levels of all data observed or modified. Raising the current level of the program counter allows computations to read data in a very flexible way "at the cost of being more limited in where they can subsequently write" [Stefan 2014]. This can lead to a problem commonly referred to as *label creep* [Sabelfeld 2003a]. To overcome this problem, these monitors make use of a special construct that allows for the evaluation of an expression in a separate context and for the reset of the program counter after that evaluation [Stefan 2011, Buiras 2014].

Recently, Buiras et al. [Buiras 2014] have presented a coarse-grained flow-sensitive information flow monitor featuring the decomposition of security labels for references into two elements: the label of the value to which the reference points and the *label of the label* of that reference. This labelling strategy gives a new perspective on the *no-sensitive-upgrade* strategy: the label of a reference can be upgraded freely as long as the label of its label remains invariant.

Monitoring Secure Information Flow in the Browser. The monitor presented in this thesis is purely dynamic and enforces the no-sensitive-upgrade discipline. Moreover, this work introduces the notions of *existence level* and *structure security level* for the labelling of JavaScript objects. However, as this monitor has been designed in order to guide a browser instrumentation and not an inlining transformation, it labels values instead of property names. For this reason, it is our opinion that the labelling abstraction presented in this chapter is better fitted for guiding the implementation of an inlining compiler that uses shadow variables and shadow properties.

Hedin and Sabelfeld [Hedin 2012] were the first to design, prove sound, and implement an information flow monitor for a realistic core of JavaScript. As their monitor is purely dynamic, it suffers from the limitations of being very conservative [Russo 2010]. To overcome these limitations, Birgisson et al. [Birgisson 2012] show how to use tests in order to boost the permissiveness of the monitor presented in [Hedin 2012]. Concretely, each time the execution of a program is blocked in order to prevent a sensitive upgrade, an upgrading instruction is added to the program in order to prevent the same error from reoccurring. Since the upgrading instruction is placed outside the sensitive context, the execution of the modified program no longer triggers the identified illegal upgrade. As a result, future executions of the modified program will not be blocked due to the same error. This methodology can be applied to the monitor presented in this chapter in order to make it less conservative.

Despite targeting JavaScript, the monitors of both Hedin [Hedin 2012] and Birgisson et al. [Birgisson 2012], as our own, do not model the reactive aspect of client-side web applications. Bohannon et al. [Bohannon 2009] presented a definition of noninterference for reactive programs such as web scripts. They further presented a runtime monitor for enforcing the proposed definition of *reactive noninterference*. Later, Bielova et al. [Bielova 2011] proposed an enforcement mechanism for reactive noninterference based on secure multi-execution [Devriese 2010], which was implemented on top of the Featherweight Firefox browser model.

Monitor-Inlining Transformations Chudnov and Naumann [Chudnov 2010] proposed an information flow monitor inlining transformation for a WHILE language, which inlines the hybrid

information flow monitor presented in [Russo 2010]. Hence, their inlining compiler includes a static analysis that estimates the set of variables updated in untaken program branches. Later, Magazinius et al. [Magazinius 2010c, Magazinius 2012] proposed the inlining of a purely dynamic information flow monitor that enforces the no-sensitive-upgrade discipline for a simple imperative language that features global functions, a let construct, and an *eval* expression that allows for dynamic code evaluation. Both compilers pair up each variable with a *shadow* variable that holds its corresponding level.

Here, we extend the techniques of [Chudnov 2010, Magazinius 2010c, Magazinius 2012] in order to handle object properties by pairing up each property with two shadow properties. The languages modelled in both [Chudnov 2010] and [Magazinius 2010c, Magazinius 2012] only feature primitive values and do not feature scope composition, where functions can be defined inside functions. In [Chudnov 2010] there are no functions and in [Magazinius 2012] every function is executed in a "clean" environment, without producing side-effects. Hence, in both [Chudnov 2010] and [Magazinius 2012], the reading effect of an expression e corresponds to the least upper bound between the levels of the variables that **explicitly occur in** e. Therefore, the instrumented code for computing the level of e is simply $\$x_1 \sqcup \cdots \sqcup \$x_n$, where $\{x_1, \cdots, x_n\}$ are the variables that explicitly occur in e and $\{\$x_1, \cdots, \$x_n\}$ are the variables that hold their corresponding levels. In Core JavaScript, as in JavaScript, this does not hold. First, one can immediately notice that expressions that feature property look-ups or function/method calls do not generally verify this property. Second, expressions may be composed of expressions that have side effects. Therefore, the level associated with the entire expression can actually be lower than the least upper bound on the levels of the variables that it includes. As an example, consider the expression $(x = y) + x$. Since $x = y$ evaluates to the value of y (besides assigning the value of y to x), the value to which the whole expression evaluates only depends on the initial value of y. Therefore, the reading effect of this expression should not take into account the initial level of x. In order to handle these two issues, an inlining transformation for JavaScript must introduce extra variables to keep track of the values and levels of intermediate expressions.

Inlining Transformations for Securing JavaScript Programs Phung et al. [Phung 2009] proposed a methodology for implementing security monitors that consist of wrapping security-critical built-in methods of JavaScript programs in order to enforce security policies. Concretely, in the context of this work, a security policy is a piece of JavaScript code specifying which method calls are to be intercepted and, for each intercepted method call, which action is to be taken. The major advantage of the methodology proposed in [Phung 2009] is that it does not require the monitored code to be re-written. Instead, it only requires a pre-step that serves to wrap security critical built-in functions with the monitoring code that checks adherence to the specified security policies. This approach suffers, however, from a range of vulnerabilities that have to do with the fact that wrapped methods are executed in the attacker's environment. Hence, the attacker can modify functions used by the wrapping functions to "bypass the policies or extract the original unwrapped methods" [Magazinius 2010b]. To overcome this issue, Magazinius et al. [Magazinius 2010b] extend the work of [Phung 2009] with a mechanism for the specification of *declarative policies* which are both easier to write and not vulnerable to the attacker's code.

4.4 Discussion

The prototype of the compiler is implemented in JavaScript and is available online at [Fragoso Santos 2014], together with a broad set of examples encompassing all of the examples provided throughout the chapter. This section discusses:

- implementation details regarding the problem of how to give security guarantees in the

presence of active attackers,

- the additional challenges introduces by implicit type coercions which are considered in the implementation of the compiler.

Untrusted Code and Native Functions Active attackers can be seen as input programs that actively try to bypass the runtime enforcement mechanism inlined by the compiler in order to trigger illegal flows without being noticed. The correctness of the instrumentation relies on the assumption that the internal variables and properties (meant for the use of the runtime enforcement mechanism) do not overlap with those of the program to be compiled. However, a malicious program may try to bypass the inlined runtime enforcement mechanism by rewriting some of the compiler's internal variables. For example, in the current implementation the security lattice is implemented as an object bound to a global variable `$lat`. Hence, a malicious program may try to modify this object in the following way: `$lat = MOST_PERMISSIVE_LATTICE`. After setting `$lat` to the most permissive lattice, the attacker code is allowed to trigger information flows otherwise forbidden. As explained in the previous section, in order to prevent this kind of malicious behaviour, the compiler acts as follows:

- It statically verifies whether or not the identifiers that explicitly appear in the code of the program are *legal*, meaning that they are not for the internal use of the inlined enforcement mechanism (e.g. `$lat`);

- It instruments property look-ups, property assignments, method calls, and property deletions to guarantee that the corresponding property is not reserved for the use of the runtime enforcement mechanism.

Another possible technique that malicious programs can explore to tamper with the internal state of the inlined runtime enforcement mechanism consists of redefining the *native functions* used by the compiler. A *native function* is a function provided by the language runtime. Interestingly, JavaScript programs are allowed to redefine *native functions*. Hence, if the compiler runtime enforcement mechanism depends on the behaviour of a native function which compiled programs are allowed to modify, the correctness of the instrumentation may be compromised. Therefore, the compiler was implemented in such a way that the inlined runtime enforcement mechanism does not rely on any kind of data/code that can be modified at runtime by compiled programs. Consider, for instance, the following program:

```
o.prop = 0;
o.hasOwnProperty = function() { false };                    (4.4)
h ? (o.prop = 1)
```

Suppose that the structure security level of the object bound to `o` is *high*. This program is illegal because updating the value of a low property in a high context constitutes a sensitive upgrade. Creating a new property in a high context is, however, allowed (because the structure security level of the object bound to `o` is *high*). Hence, the compiler must test if the object defines the property that is being set in order to decide which constraint to apply. To this end, one could use the object *hasOwnProperty* method directly, which would make the correctness of the compiler dependent on its semantics. This approach, however, would compromise the correctness of the transformation, since malicious code can redefine the *hasOwnProperty* method, thus modifying its original semantics (which is the case in this example).

Instead of using the object's *hasOwnProperty* method, the compiler uses a different one that is provided in the runtime libraries and which is not accessible to compiled code:

```
_runtime.hasOwnProperty = (function () {
    var o1 = { },
    return function(o, prop) { o1.hasOwnProperty.call(o, prop) }
})();
```
(4.5)

Type Coercions JavaScript features implicit type coercions, which are not modelled in Core JavaScript. Implicit type coercions introduce new types of implicit flows, such as the one illustrated in the example below:

```
if (private) {
    o1.toString = function () { "p1" }
} else {
    o1.toString = function () { "p2" }
}
o2.p1 = 1;
o2.p2 = 2;
public = o2[o1]
```
(4.6)

This program first sets the `toString` method of the object bound to `o1` either to a function that returns the `"p1"` or to a function that returns `"p2"` depending on the value of the *high* variable `private`. Then, it sets the properties `"p1"` and `"p2"` of the object bound to `o2` to 1 and 2, respectively. Finally, it assigns the result of the property look-up `o2[o1]` to the *low* variable `public`. Since the expression `o1` does not evaluate to a string, but to a reference, the JavaScript engine calls the method `toString` of the object bound to `o1` in order to obtain the name of the property being inspected. Since this name depends on secret information, the result of the look-up also depends on secret information.

As the semantics of Core JavaScript does not feature implicit type coercions, the inlining compiler does not take those into account. Hence, in order to guarantee the correctness of the compilation procedure for JavaScript programs, which can perform implicit type coercions, the instrumentation disallows the use of any kind of implicit type coercion. Since relying on implicit type coercions is considered bad programming practice that is error-prone and hinders maintainability [Crockford 2008], we do not find this restriction a serious shortcoming of the compiler. For example, the program above can be equivalently rewritten as follows:

```
if (h) {
    o1.toString = function () { "p1" }
} else {
    o1.toString = function () { "p2" }
}
o2.p1 = 1;
o2.p2 = 2;
public = o2[o1.toString()]
```
(4.7)

Chapter 5

Static to Hybrid Information Flow Control in Core JavaScript

Contents

This chapter presents and proves sound both a purely static type system and a hybrid type system for securing information flow in Core JavaScript. In contrast to the static type system that rejects a program due to partial information concerning the types of its sub-expressions, the hybrid type system infers a set of assertions under which that program can be securely accepted. Then, the hybrid type system inlines the inferred assertions in the original program so as to dynamically check whether these assertions are verified. If the assertions inlined in a given program are not verified at runtime, that execution of the program is caused to diverge. Hence, by deferring rejection to runtime, the hybrid type system can typecheck secure programs that purely static type systems cannot accept.

One of the major drawbacks in the development of static analyses for JavaScript is the fact that property names can be computed using string operations [Maffeis 2009]. This renders intractable the problem of deciding at the static level which property is actually being accessed in a given property look-up. Consider the following program:

$$\begin{array}{l} \texttt{o} = \{\,\}, \\ \texttt{o.secret\_prop} = \texttt{secret\_input}(), \\ \texttt{o.public\_prop} = \texttt{public\_input}(), \\ \texttt{public\_out} = \texttt{o}[\texttt{f}()] \end{array} \tag{5.1}$$

This program creates an object and assigns it to variable `o`. Then, it adds to the newly created object two properties `"secret_prop"` and `"public_prop"` which are respectively set to a secret input and a public input (specified by the user through the functions bound to the identifiers `secret_input` and `public_input`). Then, depending on the return value of the function bound

to `f`, the program assigns the value of one of these properties to a public output. In this example, deciding which is the property whose value is assigned to the public output is equivalent to predicting the dynamic behaviour of the function bound to `f`, which is, in general, undecidable. Observe that this type of issue is not exclusive of properties look-ups. It also arises in method calls, membership expressions, property assignments, and property deletions.

In order to overcome the difficulty illustrated above, previous analyses for enforcing confinement properties in JavaScript (such as that of [Maffeis 2009]) have chosen to restrict the targeted language subset, excluding property look-ups with arbitrary expressions. Here, we propose a new approach (Section 5.4), exploiting the connections between static and runtime analysis to avoid rejecting programs that are in fact secure. The key insight of our approach is that, since we aim at enforcing **termination insensitive** noninterference, the analysis may infer a set of assertions under which a program can be securely accepted and then dynamically verify whether or not these assertions hold. The original program is instrumented in such a way that if the assertions under which it is *conditionally accepted* fail to hold, its instrumentation diverges. For instance, the example presented above cannot be statically considered secure (for an arbitrary function `f`), since, in general, it is not possible to decide whether a function produces a given output. However, the following modified version of the program:

```
o = { },
o.secret_prop = secret_input(),
o.public_prop = public_input(),                                    (5.2)
_x = f(),
(_x !== "secret_prop") ? (public_out = o[_x]) : ($diverge())
```

can be securely accepted, since it diverges whenever the function bound to `f` evaluates to `"secret_prop"`. Hence, we guarantee that the potential illegal information flow never occurs. As explained in Section 4.4, JavaScript features implicit type coercions. Hence, malicious code could try to exploit this feature of the language to bypass the inlined enforcement mechanism. Therefore, in order to prevent such attacks, it suffices to instrument the generated code so that it diverges when trying to perform implicit type coercions. Since the semantics of Core JavaScript does not include implicit type coercions, both type systems presented in this chapter do not take this issue into account.

Outline This chapter is structured as follows: Section 5.1 presents the language of security types that is used in the subsequent sections for the typing of secure information flow. Section 5.3 presents the static type system, whereas Section 5.4 presents its hybrid version. Section 5.5 presents a discussion of the related work.

5.1 Security Types for Core JavaScript

This section presents the language of security types that is later used to type secure information flow in Core JavaScript.

5.1.1 Annotating Core JavaScript

In order to allow the programmer to provide additional information to the type systems, we modify the syntax of Core JavaScript. While the static type system uses the additional information for obtaining gains in precision, the hybrid type system uses it for obtaining gains in performance. Concretely, as in [Taly 2011], property look-ups, membership testing expressions, property assignments, property deletions, and method calls are annotated with a set P of the

$pf \in \mathcal{F}_\lambda$	::=	$\lambda^{\Gamma,\dot\tau} x.\,\{\mathsf{var}^{\dot\tau_1,\cdots,\dot\tau_n}\ y_1,\cdots,y_n;\ e\}$	% Parsed Function Literal
P	::=	$\{m_1,\cdots,m_n\}$	% Property Set Annotation
$e, e_0, e_1, e_2 \in \texttt{Expr}$	::=	...	
		$\mid e_0\ \mathsf{in}_i^P\ e_1$	% Membership Testing
		$\mid e_0[e_1, P]^i$	% Property Look-up
		$\mid e_0[e_1, P] = e_2$	% Property Assignment
		$\mid \mathsf{delete}^{i,P}\ e_0[e_1]$	% Property Deletion
		$\mid e_0[e_1, P](e_2)^i$	% Method Call
		$\mid \{\,\}^{\dot\tau,i}$	% Object Literal
		$\mid \mathsf{function}^{\Gamma,\dot\tau,i}(x)\{\mathsf{var}^{\dot\tau_1,\cdots,\dot\tau_n}\ y_1,\cdots,y_n;\ e\}$	% Function Literal

Table 5.1: Modified Syntax of Expressions

properties to which the corresponding expression may evaluate. This set is called a *property set annotation*. For instance, in the expression $\texttt{o}[e, \{\texttt{"foo"}, \texttt{"bar"}, \texttt{"baz"}\}]$, the property set annotation means that e always evaluates to a string equal to `"foo"`, `"bar"`, or `"baz"`. Furthermore, object literals, function literals, as well as variable declarations are annotated with their respective security types (which are explained in the next subsection) and function literals are annotated with the typing environment in which they were typed. The modified syntax is given in Table 5.1.

We say that a property set annotation P is *correct* if the expression to which it applies always evaluates to a string in P. Moreover, we say that P is *minimal* if there is no other correct P' such that $P' \subset P$. Instrumenting a program in order to dynamically check the correctness of its property set annotations is not a difficult problem. Indeed, it amounts to wrap each expression that includes a property set annotation in a conditional expression that dynamically checks whether the expression to which the property set annotation applies is contained in that property set. If that is case, the execution is allowed to go through. Otherwise, the instrumentation causes the program to diverge. It is possible to modify the specification of the hybrid type system presented in Section 5.4 so that it additionally performs the runtime checks required for verifying the correctness of property set annotations. This would, however, clutter up the presentation. Hence, we leave it implicit and we assume, in the rest of the chapter, that property set annotations are correct. But they do not have to be minimal – the property set annoation corresponding to the set `Str` of all strings is always correct. We say that two expressions e and e' are *equal up to property set annotations*, written $e \equiv e'$, if they only differ in property set annotations. Whenever a property set annotation is omitted, it is assumed to be `Str`, and the notation $e.x$ is now used as an abbreviation for $e[\mathsf{string}(x), \{\mathsf{string}(x)\}]$. For instance, `o.xpto` is used as an abbreviation for $\texttt{o}[\texttt{"xpto"}, \{\texttt{"xpto"}\}]$.

5.1.2 Syntax of Security Types

Every security type $\dot\tau \in \texttt{SType}$ is obtained by pairing up a *raw* type $\tau \in \texttt{Type}$ with a security level $\sigma \in \mathcal{L}$, where `SType` is the set of all security types and `Type` the set of all raw types. Furthermore, we denote by `SOType` the set of all object security types. The syntax of types is given in Table 5.2, where p, σ, and κ range over the sets of strings, security levels, and type variables. Given a security type τ^σ the level σ, that annotates its raw type τ, is referred to as the *external level* of the security type. The external level of a security type establishes an upper bound on the levels of the resources on which the values of that type may depend. For instance, a primitive value of type PRIM^L may only depend on *low* resources. The same applies to an object o of type $\mu\kappa.\langle p^L : \mathsf{PRIM}^H \rangle^L$. However, the value associated with o's property p may

$\tau \in \texttt{Type}$	::=	PRIM	% Prim Type
		$\mid \langle \dot{\tau}.\dot{\tau} \xrightarrow{\sigma} \dot{\tau} \rangle$	% Function Type
		$\mid \langle \kappa.\dot{\tau} \xrightarrow{\sigma} \dot{\tau} \rangle$	% Method Type
		$\mid \mu\kappa.\langle p^{\sigma} : \dot{\tau}, \cdots, p^{\sigma} : \dot{\tau}, *^{\sigma} : \dot{\tau} \rangle$	% Extensible Object Type
		$\mid \mu\kappa.\langle p^{\sigma} : \dot{\tau}, \cdots, p^{\sigma} : \dot{\tau} \rangle$	% Non-extensible Object Type
$\dot{\tau} \in \texttt{SType}$	::=	τ^{σ}	% Security Type

Table 5.2: Syntax of Types

depend on *high* resources. A typing environment $\Gamma : \texttt{Var} \rightarrow \texttt{SType}$ is a mapping from variables to types.

In contrast to class-based languages, where method types are specified inside their classes, JavaScript functions are first-class values which can be defined anywhere in the code and later assigned to properties of arbitrary objects. This creates a dependency between types for functions and types for objects, because object types include the types of their methods and function types include the type of the objects to which the keyword this is bound during execution. To break this circularity, we make use of recursive types [Amadio 1991]. However, to keep the presentation fairly simple, we restrict the occurrence of type variables to the type of the keyword this in function types. This restriction gives raise to two kinds of function types, those that use an arbitrary type as the type of the keyword this and those which instead use a type variable. In the following, we give a brief description of the possible raw types:

- The type PRIM is the type of the expressions that evaluate to primitive values.
- The type $\langle \dot{\tau}_0.\dot{\tau}_1 \xrightarrow{\sigma} \dot{\tau}_2 \rangle$ is the type of the expressions that evaluate to functions that map values of type $\dot{\tau}_1$ to values of type $\dot{\tau}_2$ and during the execution of which the keyword this is bound to an object of type $\dot{\tau}_0$. The level σ is the *writing effect* [Sabelfeld 2003a] of the functions of that type, i.e., a lower bound on the levels of the resources created/updated during their execution.
- The type $\langle \kappa.\dot{\tau} \xrightarrow{\sigma} \dot{\tau} \rangle$ coincides with the one just described, except that it is meant to be used as the type of a method. Hence, since it is specified inside the corresponding object type, the type of the keyword this is the type variable bound by that object type (see the example given in Figure 5.1).
- The type $\mu\kappa.\langle p_0^{\sigma_0} : \dot{\tau}_0, \cdots, p_n^{\sigma_n} : \dot{\tau}_n, *^{\sigma_*} : \dot{\tau}_* \rangle$ is the type of the expressions that evaluate to (references of) objects that **potentially** define properties $p_0, \cdots, p_n$, mapping each property p_i to a value of **security type** $\dot{\tau}_i$. The security type assigned to the $*$ is the *default security type* [Thiemann 2005]. The default security type of an object type $\mu\kappa.\langle p_0^{\sigma_0} : \dot{\tau}_0, \cdots, p_n^{\sigma_n} : \dot{\tau}_n, *^{\sigma_*} : \dot{\tau}_* \rangle$ is the security type of (the values assigned to) the properties of the objects of that type which are not in $\{p_0, ..., p_n\}$. Every property p_i is additionally associated with an *existence level* σ_i. The level σ_* is the *default existence level*.
- The type $\mu\kappa.\langle p_0^{\sigma_0} : \dot{\tau}_0, \cdots, p_n^{\sigma_n} : \dot{\tau}_n \rangle$ coincides with the one just described except that it does not define a default type and a default existence level. Hence, it applies to non-extensible objects. Non-extensible objects differ from extensible objects in that they are only supposed to define the properties explicitly declared in their corresponding types.

Notation Given an object security type $\dot{\tau} \in \texttt{SOType}$, we use the notation $dom(\dot{\tau})$ for the set containing the properties that explicitly appear in $\dot{\tau}$ (including $*$ if it is present), and the

$$\dot{\tau}_{contact} = \mu\kappa\ .\ \left\langle \begin{array}{l} \texttt{"fst"}^L : \mathsf{PRIM}^L,\ \texttt{"lst"}^L : \mathsf{PRIM}^L, \\ \texttt{"id"}^L : \mathsf{PRIM}^H,\ \texttt{"favourite"}^H : \mathsf{PRIM}^H, \\ \texttt{"printContact"}^L : \langle \kappa.\mathsf{PRIM}^L \xrightarrow{H} \mathsf{PRIM}^L \rangle^L, \\ \texttt{"makeFavorite"}^L : \langle \kappa.\mathsf{PRIM}^L \xrightarrow{H} \mathsf{PRIM}^L \rangle^L, \\ \texttt{"isFavorite"}^L : \langle \kappa.\mathsf{PRIM}^L \xrightarrow{H} \mathsf{PRIM}^H \rangle^L, \\ \texttt{"unFavorite"}^L : \langle \kappa.\mathsf{PRIM}^L \xrightarrow{H} \mathsf{PRIM}^H \rangle^L, \\ \texttt{"\_prot\_"}^L : \dot{\tau}_{proto\_contact} \end{array} \right\rangle^L$$

$$\dot{\tau}_{CM} = \mu\kappa\ .\ \left\langle \begin{array}{l} \texttt{"proto\_contact"}^L : \dot{\tau}_{proto\_contact}, \\ \texttt{"contact\_list"}^L : \mu\kappa.\langle *^L : \dot{\tau}_{contact} \rangle^L, \\ \texttt{"createContact"}^L : \langle \kappa.(\mathsf{PRIM}^L, \mathsf{PRIM}^L, \mathsf{PRIM}^H) \xrightarrow{L} \dot{\tau}_{contact} \rangle^L, \\ \texttt{"storeContact"}^L : \langle \kappa.(\dot{\tau}_{contact}, \mathsf{PRIM}^L) \xrightarrow{L} \dot{\tau}_{contact} \rangle^L, \\ \texttt{"getContact"}^L : \langle \kappa.(\mathsf{PRIM}^L, \mathsf{PRIM}^L) \xrightarrow{H} \dot{\tau}_{contact} \rangle^L \end{array} \right\rangle^L$$

Figure 5.1: Typing Environment for the Contact Manager - $\Gamma_{CM} = [CM \mapsto \dot{\tau}_{CM}]$

notation $*(\dot{\tau})$ for the pair $(\sigma_*, \dot{\tau}_*)$ consisting of the default existence level and the default security type of $\dot{\tau}$. Note that the fact that an object has type $\dot{\tau}$ does not mean that it defines all of the properties declared in $dom(\dot{\tau})$, but rather that it **potentially** defines the properties in $dom(\dot{\tau})$ (in which case they are mapped to values of the corresponding type).

Given a security type $\dot{\tau}$, $lev(\dot{\tau})$ denotes its external level and $\lfloor \dot{\tau} \rfloor$ its raw type. For instance, $lev(\mathsf{PRIM}^L) = L$ and $\lfloor \mathsf{PRIM}^L \rfloor = \mathsf{PRIM}$. We define $\dot{\tau}^\sigma$ as $\lfloor \dot{\tau} \rfloor^{lev(\dot{\tau}) \sqcup \sigma}$. Hence, $(\mathsf{PRIM}^L)^H = \mathsf{PRIM}^H$.

Example Figure 5.1 presents a typing environment for the Contact Manager example. We omit the specification of the type $\dot{\tau}_{proto\_contact}$ that coincides with $\dot{\tau}_{contact}$ in every property except in `"_prot_"` for which it does not define a mapping, since objects of that type are not supposed to have a prototype.[1] In the example, functions that do not modify the memory are associated with function types with *high* writing effects. This is due to the fact that the writing effect of a function is a lower bound on the levels of the resources that are updated/created during the execution of that function. Hence, when no resources are created/updated, the writing effect is the *top* security level.

Inspection of Object Types It is useful to define a function $\Rsh$: $\texttt{SOType} \times \texttt{Str} \to \mathcal{L} \times \texttt{SType}$ that receives as input an object security type $\dot{\tau}$ and a string p and outputs a pair consisting of the existence level and the security type with which $\dot{\tau}$ associates p:

$$\Rsh\ (\dot{\tau}, p) = \begin{cases} (\sigma_i, \{\dot{\tau}/\kappa\}\dot{\tau}_p) & \text{if } \dot{\tau} = \mu\kappa.\langle \cdots, p^{\sigma_i} : \dot{\tau}_p, \cdots \rangle^\sigma \\ (\sigma_*, \{\dot{\tau}/\kappa\}\dot{\tau}_*) & \text{if } \dot{\tau} = \mu\kappa.\langle \cdots, *^{\sigma_*} : \dot{\tau}_*, \cdots \rangle^\sigma \ \wedge\ p \notin dom(\dot{\tau}) \end{cases} \tag{5.3}$$

where $\{\dot{\tau}_0/\kappa\}\dot{\tau}_1$ denotes the capture-avoiding substitution of κ for $\dot{\tau}_0$ in $\dot{\tau}_1$. In the following, given a pair $lt = (\sigma, \dot{\tau})$ consisting of a security level and a security type, we use $\pi_{\texttt{lev}}(lt)$ to denote σ and $\pi_{\texttt{type}}(lt)$ to denote $\dot{\tau}$.

Interestingly, object types can be interpreted as a typing environments. Concretely, given an object security type $\dot{\tau} \in \texttt{SOType}$, we define its corresponding typing environment, $\Gamma_{\dot{\tau}} : \texttt{Var} \to \texttt{SType}$, as the function that maps each identifier whose name corresponds to a string $p \in dom(\dot{\tau})$ to the security type with which that type associates p – $\pi_{\texttt{type}}(\Rsh\ (\dot{\tau}, p))$. Formally: $\Gamma_{\dot{\tau}}(x) =$

[1] Note that in real JavaScript every object has an implicit prototype: `Object.prototype`.

$\pi_{\texttt{type}}(\Rsh\ (\dot{\tau}, \mathsf{string}(x)))$. Conversely, given a typing environment Γ, $\dot{\tau}_\Gamma$ denotes the object security type that matches Γ. Formally, given a typing environment Γ such that $dom(\Gamma) = \{y_1, \cdots, y_n\}$, $\dot{\tau}_\Gamma$ is defined as $\mu\kappa.\langle p_1^\perp : \dot{\tau}_1, \cdots, p_n^\perp : \dot{\tau}_n\rangle$ where $p_i = \mathsf{string}(y_i)$ and $\dot{\tau}_i = \Gamma(y_i)$ for $i = 1, \cdots, n$.

5.1.2.1 Restricting the Syntax of Security Types for Objects

In order to guarantee the soundness of the proposed type systems, one must impose some restrictions on the syntax of object security types. This subsection describes these restrictions.

First, we require the existence level of a property to be lower than or equal to the level that annotates its corresponding security type. This restriction forbids the specification of an object security type that associates an invisible property with a visible value. Formally, given an object security type $\dot{\tau}$ and a property $p \in dom(\dot{\tau})$, it must be the case that:

$$\pi_{\texttt{lev}}(\Rsh\ (\dot{\tau}, p)) \sqsubseteq lev(\pi_{\texttt{type}}(\Rsh\ (\dot{\tau}, p))) \tag{5.4}$$

Second, we require the security level that annotates an object type to be higher than or equal to the level that annotates the type of its prototype. This constraint is meant to prevent leaks *via* prototype mutations. If the level of the prototype of an object o is *high*, then the prototype of o is allowed to change in a *high* context. However, such changes remain invisible to a *low* observer, because the level of o is itself *high*, meaning that a *low* observer can never see any of the contents of o. Formally, given an object security type $\dot{\tau}$, it must be the case that:

$$lev(\pi_{\texttt{type}}(\Rsh\ (\dot{\tau}, \texttt{"\_prot\_"}))) \sqsubseteq lev(\dot{\tau}) \tag{5.5}$$

The final restriction concerning the syntax of security types for objects has to do with the relation between the type of an object and the type of its prototype. An important aspect of object types is that they must reflect the whole prototype-chain accessible through the corresponding objects. Hence, in the Contact Manager example, the security type assigned to contact objects also includes the methods that the corresponding prototype implements. Since every object type must reflect the whole prototype-chain accessible through the corresponding objects, not all types can be used as the *type of the prototype* for the objects of a given type.

Consider, for instance, an object o_0 of type $\dot{\tau}_0 = \mu\kappa.\langle \texttt{"propA"}^L : \mathsf{PRIM}^L, \texttt{"\_prot\_"}^L : \_\rangle^L$ and an object o_1 of type $\dot{\tau}_1 = \mu\kappa.\langle \texttt{"propA"}^L : \mu\kappa.\langle *^L : \mathsf{PRIM}^L\rangle^L\rangle^L$. Suppose we set $\dot{\tau}_1$ as the type of the prototype in $\dot{\tau}_0$. Then, the look-up of `"propA"` in o_0 may yield two different types of values (besides undefined, if neither o_0 nor o_1 defines `"propA"`). It yields a value of type PRIM^L when object o_0 defines `"propA"` and an object of type $\mu\kappa.\langle *^L : \mathsf{PRIM}^L\rangle^L$ when o_0 does not define `"propA"` and o_1 defines `"propA"`. In order to overcome this problem, we restrict what types can be legally used for the prototype of a given object type. We say that $\dot{\tau}_1$ is a *consistent prototype type* for $\dot{\tau}_0$, written $\dot{\tau}_0 \preceq_{proto} \dot{\tau}_1$, if $\dot{\tau}_1$ does not define a default type and both types coincide in the domain of $\dot{\tau}_1$ (with the exception of the property `"_prot_"`). Definition 5.1 formalises this notion.

Definition 5.1 (Consistent Prototypes)**.** *We say that $\dot{\tau}_1$ is a consistent prototype type for $\dot{\tau}_0$, written $\dot{\tau}_0 \preceq_{proto} \dot{\tau}_1$, if and only if:*

- $* \notin dom(\dot{\tau}_1)$,
- $dom(\dot{\tau}_1) \subseteq dom(\dot{\tau}_0)$,
- $\forall_{p \in dom(\dot{\tau}_1) \setminus \{\texttt{"\_prot\_"}\}}\ \Rsh\ (\dot{\tau}_0, p) = \Rsh\ (\dot{\tau}_1, p)$.

5.1.2.2 Subtyping Security Types

In order to type expressions that either result from the combination of subexpressions with different types, or whose evaluation may yield values of different types (for instance, a property look-up with an imprecise property set annotation), both the type systems presented in the following sections make use of an ordering on security types, called *subtyping relation*. Intuitively, a security type $\dot{\tau}_0$ is a subtype of another security type $\dot{\tau}_1$, if the use of an expression of type $\dot{\tau}_0$ is **secure** whenever the use of an expression of type $\dot{\tau}_1$ is secure. The ordering $\sqsubseteq$ on security levels induces a simple ordering $\preceq$ on security types: $\dot{\tau}_0 \preceq \dot{\tau}_1$ *iff* $lev(\dot{\tau}_0) \sqsubseteq lev(\dot{\tau}_1)$ and $\lfloor \dot{\tau}_0 \rfloor \equiv \lfloor \dot{\tau}_1 \rfloor$, where $\equiv$ stands for syntactic equality.

As is the case of references in ML [Pottier 2002], every two object security types in the subtyping relation need to have the same corresponding raw type, because, while property look-ups and membership testing expressions are *covariant* with the type of the property being inspected, property assignments and property deletions are *contravariant*. Consider, for instance, an object of type $\dot{\tau}_0 = \mu\kappa.\langle \texttt{"propA"}^L : \textsf{PRIM}^L \rangle^L$ bound to an identifier `x` and an object of type $\dot{\tau}_1 = \mu\kappa.\langle \texttt{"propA"}^L : \textsf{PRIM}^H \rangle^L$ bound to and identifier `y`. The following two expressions illustrate why raw object types need to be invariante:

- If we let $\dot{\tau}_0 \preceq \dot{\tau}_1$, the expression $\texttt{y} = \texttt{x}, \texttt{y.propA} = \texttt{h}$, which assigns an invisible value to a visible property, would be typable.
- Conversely, if we let $\dot{\tau}_1 \preceq \dot{\tau}_0$, the expression $\texttt{x} = \texttt{y}, \texttt{l} = \texttt{x.propA}$, which assigns an invisible value to a visible variable, would be typable.

Given a raw type τ, the set $\{\dot{\tau} \mid \lfloor \dot{\tau} \rfloor \equiv \tau\}$ of its corresponding security types forms a lattice (when ordered by $\preceq$). The corresponding *lub* and *glb* $\curlyvee, \curlywedge : \texttt{SType} \times \texttt{SType} \to \texttt{SType}$ are defined as follows: $\dot{\tau}_0 \mathbin{\curlywedge}^{\curlyvee} \dot{\tau}_1 = \dot{\tau} \Leftrightarrow \lfloor \dot{\tau} \rfloor \equiv \lfloor \dot{\tau}_0 \rfloor \equiv \lfloor \dot{\tau}_1 \rfloor \wedge lev(\dot{\tau}) = lev(\dot{\tau}_0) \overset{\sqcup}{\sqcap} lev(\dot{\tau}_1)$. Using the notions of *lub* and *glb* between security types, we extend $\Rsh$ to arbitrary sets of properties in the two following ways $\Rsh_{\uparrow}, \Rsh_{\downarrow} : \texttt{SOType} \times 2^{\texttt{Str}} \to \mathcal{L} \times \texttt{SType}$:

$$\Rsh_{\uparrow}(\dot{\tau}, P) = (\sqcup\{\hat{\sigma} \mid p \in P \wedge \hat{\sigma} = \pi_{\texttt{lev}}(\Rsh(\dot{\tau}, p))\}, \curlyvee\{\dot{\tau}' \mid p \in P \wedge \dot{\tau}' = \pi_{\texttt{type}}(\Rsh(\dot{\tau}, p))\}) \quad (5.6)$$

$$\Rsh_{\downarrow}(\dot{\tau}, P) = (\sqcap\{\hat{\sigma} \mid p \in P \wedge \hat{\sigma} = \pi_{\texttt{lev}}(\Rsh(\dot{\tau}, p))\}, \curlywedge\{\dot{\tau}' \mid p \in P \wedge \dot{\tau}' = \pi_{\texttt{type}}(\Rsh(\dot{\tau}, p))\}) \quad (5.7)$$

While $\Rsh_{\uparrow}$ is used for the typing of property look-ups, membership testing expressions, and method calls (which are covariant with the type of the corresponding property), $\Rsh_{\downarrow}$ is used for the typing of property assignments and property deletions (which are contravariant with the type of the corresponding property).

5.1.3 Well-Typed Memories

In order to reason about the types of the objects in memory, we have to extend the semantics of Core JavaScript, defined in Section 2.4, with *type-based labellings* that serve to record the types of the objects created at runtime, which include the types of the function literals dynamically evaluated. Hence, the augmented transitions of the big-step semantics for Core JavaScript have the form:

$$r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu', \Sigma', v \rangle \quad (5.8)$$

where Σ and Σ' are initial and final *type-based labellings* respectively, while the remaining elements keep their original meaning. A *type-based labelling* is a function $\Sigma : \texttt{Ref} \to \texttt{SType}$ mapping

$$\textsc{Function Literal}\quad \frac{r' = \mathsf{fresh}(\mu) \qquad \Sigma' = \Sigma\,[r' \mapsto \dot{\tau}] \qquad \mu' = \mu\left[r' \mapsto \left[\texttt{"@fscope"} \mapsto r, \texttt{"@code"} \mapsto \lambda^{\Gamma,\dot{\tau}} x.\left\{\mathsf{var}^{\dot{\tau}_1,\cdots,\dot{\tau}_n}\ y_1,\cdots,y_n;\ e\right\}\right]\right]}{r \vdash \langle \mu, \Sigma, \mathsf{function}^{\Gamma,\dot{\tau},i}(x)\{\mathsf{var}^{\dot{\tau}_1,\cdots,\dot{\tau}_n}\ y_1,\cdots,y_n;\ e\}\rangle \Downarrow \langle \mu', \Sigma', r'\rangle}$$

$$\textsc{Object Literal}\quad \frac{r' = \mathsf{fresh}(\mu) \qquad \Sigma' = \Sigma\,[r' \mapsto \dot{\tau}] \qquad \mu' = \mu\,[r' \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}]]}{r \vdash \langle \mu, \Sigma, \{\,\}^{\dot{\tau},i}\rangle \Downarrow \langle \mu', \Sigma', r'\rangle}$$

Figure 5.2: A Big-Step Semantics for Core JavaScript Extended with Type-based Labellings

references to security types. Upon the evaluation of a function/object literal of type $\dot{\tau}$, the semantics extends the current labelling Σ with a new mapping from the newly created reference to its corresponding type. The two unique rules that directly interact with type-based labellings are [Function Literal] and [Object Literal]. These rules are presented in Figure 5.2. The only difference between these two rules and their original counterparts is that these rules have to extend the current type-based labelling with a new mapping from the newly allocated reference to the type of its corresponding object.

Another important difference between the adapted semantics of Core JavaScript used in this chapter and the one introduced in Chapter 2 is that, here, parsed function literals in memory are assumed to be annotated with the typing environment in which they were typed. Accordingly, we assume the existence of a semantic function tenv that, given a parsed function literal, outputs the typing environment with which it is annotated. For instance, given a memory μ and a reference r pointing to a function object in μ, $\mathsf{tenv}(\mu(r \cdot \texttt{"@code"}))$ is the typing environment that annotates the function literal associated with the function object pointed to by r. It is important to emphasise that this is just a device for the proofs and not a feature of the enforcement mechanism. In other words, these typing environments are not used by the semantics.

We can now introduce the definition of *well-typed memory*. Informally, one can say that a memory is *well-typed* by a given type-based labelling Σ, if the types given by Σ to the objects in memory "match" the objects with which they are associated. In the same way, a scope-chain is *well-typed* by a given typing environment Γ and type-based labelling Σ, if the types assigned by Γ to the identifiers in that scope match their corresponding values. Definition 5.2 establishes the notion of *well-typed scope-chain*, whereas Definition 5.3 gives the notion of *well-typed memory*.

In order to simplify the specification of the following two definitions, it is useful to introduce the notion of *extended labelling to primitive values*. Given a labelling $\Sigma : \texttt{Ref} \rightarrow \texttt{SType}$, its extension to primitive values $\bar{\Sigma} : \texttt{Ref} \cup \texttt{Prim} \rightarrow \texttt{SType}$ is defined as follows:

$$\bar{\Sigma}(v) = \begin{cases} \Sigma(v) & \text{if } v \in \texttt{Ref} \\ \mathsf{PRIM}^{\perp} & \text{otherwise} \end{cases} \tag{5.9}$$

Definition 5.2 (Well-typed Scope-Chain)**.** *Given a memory μ, a scope reference r, a typing environment Γ, and a type-based labelling Σ, we say that the scope-chain stored in μ that starts in r is well-typed by Γ and Σ if for every variable $x \in dom(\Gamma)$ for which there is a reference r' such that $r' = \mathsf{Scope}(\mu, r, x)$ and $r' \neq \mathsf{null}$, it follows that: $\bar{\Sigma}(\mu(r_x \cdot m_x)) \preceq \Gamma(x)$, where $m_x = \mathsf{string}(x)$.*

Definition 5.3 (Well-Typed Memory)**.** *A memory μ is well-typed by Σ, if:*

1. *every reference pointing to a non-internal object in μ is in the domain of Σ,*

2. *every reference r_f pointing to a funtion object in μ is mapped by Σ to a function type $\dot{\tau}$, which correctly types the body of the corresponding function in its annotated typing environment ($\Gamma = \mathsf{tenv}(\mu(r_f \cdot \texttt{"@code"}))$), and the corresponding scope-chain is well-typed by Γ and Σ,*

3. *for every reference $r \in dom(\Sigma)$ and property $p \in dom(\mu(r))$, it holds that:*

$$\Sigma(\mu(r \cdot p)) \preceq \pi_{\mathit{type}}(\Gamma\!\!\uparrow (\Sigma(r), p))$$

5.2 The Attacker Model and the Meaning of Security Types

Since in this chapter resources are labelled using type-based labellings instead of dynamic labellings, the low-equality definition must be adjusted to type-based labellings. Informally, when considering a memory well-typed by a type-based labelling Σ, an attacker at level σ can see:

1. a reference r as well as the type of the object to which it points, provided that the external level of that type is lower than or equal to σ (formally, $lev(\Sigma(r)) \sqsubseteq \sigma$),

2. the existence of a property p in an object pointed to by a **visible** reference r, provided that the type of that object associates p with an existence level lower than or equal to σ (formally, $\pi_{\mathtt{lev}}(\Gamma\!\!\uparrow (\Sigma(r), p)) \sqcup lev(\Sigma(r)) \sqsubseteq \sigma$),

3. the value of a property p in an object pointed to by a **visible** reference r, provided that the type of that object associates p with a security type whose external level is lower than or equal to σ (formally, $lev(\pi_{\mathtt{type}}(\Gamma\!\!\uparrow (\Sigma(r), p))) \sqcup lev(\Sigma(r)) \sqsubseteq \sigma$),

4. the code of a function object pointed to by a reference r, provided that the external level of the type of that object is lower than or equal to σ (formally, $lev(\Sigma(r)) \sqsubseteq \sigma$).

Since every function object in memory is associated with the scope object that was active at the time of its evaluation, the low-equality must also take into account the scope-chains that are stored in memory. To this end, Definition 5.4 extends the notion of low-projection and low-equality to scope-chains. Informally, given a labelling Γ, a memory μ, and a scope reference r, an attacker at level σ can see:

- the value of a variable x in the domain of Γ, provided that there is an object in the scope-chain that starts with $\mu(r)$ that defines a binding for x and that x is mapped by Γ to a security type whose external level is lower than or equal to σ (formally, $lev(\Gamma(x)) \sqsubseteq \sigma$).

Definition 5.4 (Low-Projection and Low-Equality for Scope-Chains)**.** *The low-projection at security level σ of the scope-chain labelled by Γ that starts with the scope object pointed to by r in memory μ is defined as follows:*

$$(\mu, r)\ {\restriction}^{\Gamma,\sigma} = \{(x, r_x, \mu(r_x \cdot m_x)) \mid x \in dom(\Gamma)\ \wedge\ lev(\Gamma(x))) \sqsubseteq \sigma\ \wedge\ r_x = \mathsf{Scope}(\mu, r, x) \wedge \\ \wedge\ r_x \neq \mathsf{null}\ \wedge\ m_x = \mathsf{string}(x)\}$$

We say that the scope-chains starting with two objects pointed to by the same reference r in two memories μ_0 and μ_1 are low-equal at level σ w.r.t. Γ, written $\Gamma, r \Vdash \mu_0 \sim_\sigma \mu_1$, if $(\mu_0, r)\ {\restriction}^{\Gamma,\sigma} = (\mu_1, r)\ {\restriction}^{\Gamma,\sigma}$.

Definition 5.5 (Low-Projection and Low-Equality for Memories)**.** *The low-projection of a memory μ w.r.t. a security level σ and a type-based labelling Σ is given by:*

$$\begin{aligned}\mu \restriction^{\Sigma,\sigma} = \ & \{(r, \Sigma(r)) \mid lev(\Sigma(r)) \sqsubseteq \sigma\} \\ & \cup \ \{(r,p) \mid \ \pi_{\mathtt{lev}}(\vec{\ell} \ (\Sigma(r),p)) \sqcup lev(\Sigma(r)) \sqsubseteq \sigma \ \wedge \ p \in dom(\mu(r))\} \\ & \cup \ \{(r,p,v) \mid \ lev(\pi_{\mathtt{type}}(\vec{\ell} \ (\Sigma(r),p))) \sqcup lev(\Sigma(r)) \sqsubseteq \sigma \ \wedge \ v = \mu(r \cdot p)\} \\ & \cup \ \{\big(r, f, r_s, (\mu, r_s) \restriction^{\Gamma,\sigma}\big) \mid lev(\Sigma(r)) \sqsubseteq \sigma \ \wedge \ f = \mu(r \cdot \texttt{"@code"}) \ \wedge \ \Gamma = \mathsf{tenv}(f) \\ & \qquad \wedge \ r_s = \mu(r \cdot \texttt{"@fscope"})\}\end{aligned}$$

Two memories μ_0 and μ_1, respectively typed by Σ_0 and Σ_1 are said to be low-equal at security level σ, written $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$ if they coincide in their respective low-projections, $\mu_0 \restriction^{\Sigma_0,\sigma} = \mu_1 \restriction^{\Sigma_1,\sigma}$.

5.2.1 Noninterference for Typed Programs

Informally, a program is *noninterferent* if its execution in two low-equal memories always produces two low-equal memories. Hence, an attacker cannot use a noninterferent program as a means to disclose the confidential contents of a memory. In the following, given a typing environment Γ, we say that a type-based labelling Σ is consistent with Γ if $\Sigma(\#glob) = \dot{\tau}_\Gamma$, meaning that the type of the global $\dot{\tau}_{glob}$ matches the typing environment.

Definition 5.6 (Noninterference)**.** *An expression e is said be noninterferent with respect to a typing environment Γ, written $\mathbf{NI}(e, \Gamma)$, if for any two memories μ and μ', type-based labellings Σ and Σ', and security level $\sigma \in \mathcal{L}$ such that:*

- *μ is well-typed by Σ and μ' is well-typed by Σ',*
- *Σ and Σ' are consistent with Γ,*
- *$\#glob \vdash \langle \mu, \Sigma, e\rangle \Downarrow \langle \mu_f, \Sigma_f, v\rangle$,*
- *$\#glob \vdash \langle \mu', \Sigma', e\rangle \Downarrow \langle \mu'_f, \Sigma'_f, v'\rangle$, and*
- *$\mu, \Sigma \sim_\sigma \mu', \Sigma'$;*

It holds that: $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$.

The definition of noninterference is standard except for the requirement that the typing environment be consistent with the type of the global object. Furthermore, the initial memories are assumed to be *well-typed*, meaning that the types of the references in memory must "match" their corresponding values. For simplicity, the definition of noninterference does not impose any restriction on the generated outputs. This does not constitute a problem, since any expression e that produces a *high* output can be trivially re-written as $\mathtt{h} = e, \mathsf{null}$, for an arbitrary *high* variable $\mathtt{h}$.

5.3 Static Information Flow Control in Core JavaScript

We now present a static type system for securing information flow in Core JavaScript. The rules, presented in Figure 5.3, use typing judgements of the form $\Gamma, \sigma_{pc} \vdash e : \dot{\tau}$, where:

- Γ is the typing environment,
- σ_{pc} the *context level*, that is, a lower bound on the levels of the resources that can be updated/created when e is evaluated,

$$\textsc{Value}\quad \Gamma, \sigma_{pc} \vdash v : \textsf{PRIM}^{\perp} \qquad \textsc{This}\quad \Gamma, \sigma_{pc} \vdash \textsf{this} : \Gamma(\textsf{this}) \qquad \textsc{Var}\quad \Gamma, \sigma_{pc} \vdash x : \Gamma(x) \qquad \textsc{Object Literal}\quad \frac{\sigma_{pc} \sqsubseteq lev(\dot{\tau})}{\Gamma, \sigma_{pc} \vdash \{\}^{\dot{\tau}} : \dot{\tau}}$$

$$\textsc{Binary Operation}\quad \frac{\Gamma, \sigma_{pc} \vdash e_i : \dot{\tau}_i \quad \dot{\tau} = \dot{\tau}_0 \curlyvee \dot{\tau}_1}{\Gamma, \sigma_{pc} \vdash e_0 \ \textsf{op}\ e_1 : \dot{\tau}} \qquad \textsc{Variable Assignment}\quad \frac{\Gamma, \sigma_{pc} \vdash e : \dot{\tau} \quad \dot{\tau}^{\sigma_{pc}} \preceq \Gamma(x)}{\Gamma, \sigma_{pc} \vdash x = e : \dot{\tau}}$$

$$\textsc{Property Look-up}\quad \frac{\forall_{i=0,1} \cdot \Gamma, \sigma_{pc} \vdash e_i : \dot{\tau}_i \quad \dot{\tau} = \pi_{\texttt{type}}(\Uparrow_{\uparrow} (\dot{\tau}_0, P)) \quad \sigma = lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1)}{\Gamma, \sigma_{pc} \vdash e_0[e_1, P] : \dot{\tau}^{\sigma}} \qquad \textsc{Membership Testing}\quad \frac{\forall_{i=0,1} \cdot \Gamma, \sigma_{pc} \vdash e_i : \dot{\tau}_i \quad \sigma = \pi_{\texttt{lev}}(\Uparrow_{\uparrow} (\dot{\tau}_1, P)) \sqcup lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1)}{\Gamma, \sigma_{pc} \vdash e_0 \ \textsf{in}^{P}\ e_1 : \textsf{PRIM}^{\sigma}}$$

$$\textsc{Property Assignment}\quad \frac{\forall_{i=0,1,2} \cdot \Gamma, \sigma_{pc} \vdash e_i : \dot{\tau}_i \quad (\sigma, \dot{\tau}) = \Uparrow_{\downarrow} (\dot{\tau}_0, P) \quad \dot{\tau}_2 \preceq \dot{\tau} \quad \sigma_{pc} \sqcup lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqsubseteq \sigma}{\Gamma, \sigma_{pc} \vdash e_0[e_1, P] = e_2 : \dot{\tau}_2}$$

$$\textsc{Property Deletion}\quad \frac{\forall_{i=0,1} \cdot \Gamma, \sigma_{pc} \vdash e_i : \dot{\tau}_i \quad \sigma = \pi_{\texttt{lev}}(\Uparrow_{\downarrow} (\dot{\tau}, P)) \quad \sigma_{pc} \sqcup lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqsubseteq \sigma}{\Gamma, \sigma_{pc} \vdash \textsf{delete}^{P}\ e_0[e_1] : \textsf{PRIM}^{\perp}}$$

$$\textsc{Function Call}\quad \frac{\Gamma, \sigma_{pc} \vdash e_0 : \langle \dot{\tau}_0'.\dot{\tau}_1' \xrightarrow{\hat{\sigma}} \dot{\tau}_2' \rangle^{\hat{\sigma}'} \quad \Gamma, \sigma_{pc} \vdash e_1 : \dot{\tau}_1 \quad \sigma = \sigma_{pc} \sqcup \hat{\sigma}' \quad \sigma \sqsubseteq \hat{\sigma} \quad \dot{\tau}^{\sigma}_{global} \preceq \dot{\tau}_0' \quad \dot{\tau}_1^{\sigma} \preceq \dot{\tau}_1'}{\Gamma, \sigma_{pc} \vdash e_0(e_1) : (\dot{\tau}_2')^{\sigma}}$$

$$\textsc{Method Call}\quad \frac{\forall_{i=0,1,2} \cdot \Gamma, \sigma_{pc} \vdash e_i : \dot{\tau}_i \quad \langle \dot{\tau}_0'.\dot{\tau}_1' \xrightarrow{\hat{\sigma}} \dot{\tau}_2' \rangle^{\hat{\sigma}'} = \pi_{\texttt{type}}(\Uparrow_{\uparrow} (\dot{\tau}_0, P)) \quad \sigma = \sigma_{pc} \sqcup \hat{\sigma}' \sqcup lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \quad \sigma \sqsubseteq \hat{\sigma} \quad \dot{\tau}_0^{\sigma} \preceq \dot{\tau}_0' \quad \dot{\tau}_2^{\sigma} \preceq \dot{\tau}_1'}{\Gamma, \sigma_{pc} \vdash e_0[e_1, P](e_2) : (\dot{\tau}_2')^{\sigma}}$$

$$\textsc{Conditional Expression}\quad \frac{\Gamma, \sigma_{pc} \vdash e_0 : \dot{\tau}_0 \quad \sigma'_{pc} = \sigma_{pc} \sqcup lev(\dot{\tau}_0) \quad \forall_{i=1,2} \cdot \Gamma, \sigma'_{pc} \vdash e_i : \dot{\tau}_i \quad \dot{\tau} = \dot{\tau}_1 \curlyvee \dot{\tau}_2}{\Gamma, \sigma_{pc} \vdash e_0\ ?\ (e_1) : (e_2) : \dot{\tau}^{lev(\dot{\tau}_0)}} \qquad \textsc{Sequence}\quad \frac{\forall_{i=0,1} \cdot \Gamma, \sigma_{pc} \vdash e_i : \dot{\tau}_i}{\Gamma, \sigma_{pc} \vdash e_0, e_1 : \dot{\tau}_1}$$

$$\textsc{Function Literal}\quad \frac{\dot{\tau} = \langle \dot{\tau}_0'.\dot{\tau}_1' \xrightarrow{\hat{\sigma}} \dot{\tau}_2' \rangle^{\sigma} \quad \sigma_{pc} \sqcup \sigma \sqsubseteq \hat{\sigma} \quad \Gamma\,[\textsf{this} \mapsto \dot{\tau}_0', x \mapsto \dot{\tau}_1', y_1 \mapsto \dot{\tau}_1, \cdots, y_n \mapsto \dot{\tau}_n], \hat{\sigma} \vdash e : \dot{\tau}_2'}{\Gamma, \sigma_{pc} \vdash \textsf{function}^{\dot{\tau}, \Gamma}(x)\{\textsf{var}^{\dot{\tau}_1, \cdots, \dot{\tau}_n}\ y_1, \cdots, y_n;\ e\} : \dot{\tau}}$$

Figure 5.3: Typing Secure Information Flow in Core JavaScript

- e is the expression to be typed, and
- $\dot{\tau}$ the type that is assigned to it.

In the following, we give a brief description of the rules that compose the type system.

- [Value] A literal value is given the type $\textsf{PRIM}^{\perp}$ as it always evaluates to primitive value and it does not disclose any secret contents.
- [This] The keyword this is given the type of the object that is to be bound to the keyword this at runtime. This type is associated with the identifier this in the current typing environment.
- [Variable] A variable x is given the type with which it is associated in the current typing environment.
- [Object Literal] An object literal is given the type with which it is annotated.

Typing Environment:

$$\Gamma(\texttt{h1}) = \Gamma(\texttt{h2}) = \mathsf{PRIM}^H \quad \Gamma(\texttt{l1}) = \Gamma(\texttt{l2}) = \mathsf{PRIM}^L$$
$$\Gamma(\texttt{o1}) = \mu\kappa.\langle \texttt{"p1"}^L : \mathsf{PRIM}^H, \texttt{"p2"}^L : \mathsf{PRIM}^L, *^L : \mathsf{PRIM}^L \rangle^L$$
$$\Gamma(\texttt{o2}) = \mu\kappa.\langle \texttt{"q1"}^L : \mathsf{PRIM}^L, \texttt{"q2"}^H : \mathsf{PRIM}^H, *^L : \mathsf{PRIM}^H \rangle^L$$

Examples:

$\texttt{l1} = \texttt{o1}[\texttt{l2}, \{\texttt{"p2"}\}]$	**TYPED**
$\texttt{o1}[\texttt{l2}, \{\texttt{"p1"}, \texttt{"p3"}\}]$? $(\texttt{o2}[\texttt{l1}, \{\texttt{"q2"}\}] = 0)$	**TYPED**
$\texttt{l1} = \texttt{l2}\ \mathsf{in}^{\mathtt{Str}}\ \texttt{o2}$	**NOT TYPED**
$\texttt{o1}[\texttt{l2}, \{\texttt{"p1"}, \texttt{"p3"}\}] = \texttt{o2}[\texttt{l1}, \{\texttt{"q2"}, \texttt{"q3"}\}]$	**NOT TYPED**

Table 5.3: Examples of Programs Accepted/Rejected by the Static Type System

- [Binary Operation] A binary operation is typed with the least upper bound $\curlyvee$ between the types of its subexpressions.
- [Variable Assignment] A variable assignment $x = e$ is typed with the type of e provided that this type is a subtype of the type of x in the current typing environment and that the level of the program counter is lower than or equal to the external level of the type of x. While the first constraint prevents the assignment of a secret value to a public variable, the second constraint prevents public variables to be updated inside secret contexts (thereby preventing the execution of assignments which encode implicit flows).
- [Property Look-up] In order to type a property look-up $e_0[e_1, P]$, the type system first computes the security type $\dot{\tau}_0$ of e_0 and the security type $\dot{\tau}_1$ of e_1. Then, the type system computes the *lub* between the types with which $\dot{\tau}_0$ associates the properties in P, thereby obtaining the security type $\dot{\tau}$ (remark that this type may not exist). To account for possible implicit flows, the external level of the type given to the whole expression must be higher than or equal to the *lub* between the external levels of $\dot{\tau}_0$ and $\dot{\tau}_1$ – $\sigma = lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1)$. Hence, the whole expression is typed with $\dot{\tau}^\sigma$.
- [Membership Testing] In order to type a membership testing expression $e_0\ \mathsf{in}^P\ e_1$, the type system first computes the security type $\dot{\tau}_0$ of e_0 and the security type $\dot{\tau}_1$ of e_1. Then, the type system computes the *lub* between the existence levels with which $\dot{\tau}_1$ associates the properties in P, thereby obtaining the security level σ (remark that this level always exists). To account for possible implicit flows, the external level of the type given to the whole expression must be higher than or equal to the *lub* between the external levels of $\dot{\tau}_0$ and $\dot{\tau}_1$. Hence, the whole expression is typed with $\mathsf{PRIM}^{\sigma \sqcup lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1)}$. The raw type is PRIM because a membership testing expression always evaluates to a boolean value.
- [Property Assignment] In order to type a property assignment $e_0[e_1, P] = e_2$, the type system first computes the security types $\dot{\tau}_0$, $\dot{\tau}_1$, and $\dot{\tau}_2$ of e_0, e_1, and e_2, respectively. Then, the type system computes the *glb* between the existence levels as well as the security types with which $\dot{\tau}_0$ associates the properties in P, thereby obtaining a security level σ and a security type $\dot{\tau}$. The type system subsequently checks whether $\dot{\tau}_2$ is a subtype of $\dot{\tau}$. This constraint prevents the *explicit flow* resulting from the assignment of a *high* value to a

low property. Then, the type system checks whether the external levels of $\dot{\tau}_0$ and $\dot{\tau}$ as well as the level of the program counter are lower than or equal to σ. This constraint prevents a program from updating/creating a property with a *low* existence level/value level depending on secret information.

- [Property Deletion] In order to type a property deletion $\mathsf{delete}^P\ e_0[e_1]$, the type system first computes the security types $\dot{\tau}_0$ and $\dot{\tau}_1$ of e_0 and e_1, respectively. Then, it computes the *glb* between the existence levels with which $\dot{\tau}_0$ associates the properties in P, thereby obtaining the security level σ. Finally, the type system checks whether the *lub* between the external levels of $\dot{\tau}_0$ and $\dot{\tau}_1$ as well as the level of the program counter are lower than or equal to σ. This constraint prevents the deletion of a visible property depending on secret information. The whole expression is typed with $\mathsf{PRIM}^{\perp}$, because the evaluation of the expression always produces a boolean value and does not reveal any secret information.

- [Function Call] In order to type a function call, the type system first types its two subexpressions, thereby obtaining two types – $\dot{\tau}_0$ and $\dot{\tau}_1$. Then, it computes the least upper bound between the level that annotates $\dot{\tau}_0$ and the current level of the program counter σ_{pc} – σ. This level can be seen as an upper bound on the levels of the resources that determine at runtime the function to which e_0 evaluates. The type system, then, checks whether σ is lower than or equal to the writing effect of the function type $\dot{\tau}_0$. This constraint prevents the calling of a function that creates/updates *low* memory depending on *high* values. Finally, the type system checks whether the type $\dot{\tau}_{global}$ of the global objec and the type $\dot{\tau}_1$ of the function argument match the type $\dot{\tau}'_0$ of the keyword this and of the type $\dot{\tau}'_1$ of the function formal parameter. The whole function call is typed with the return type of the function type $\dot{\tau}'_2$. However, in order to account for possible implicit flows, its external level must be upgraded so that it is higher than or equal to σ.

- [Method Call] In order to type a method call $e_0[e_1](e_2)$, the type system first types its three subexpressions, thereby obtaining three types – $\dot{\tau}_0$, $\dot{\tau}_1$, and $\dot{\tau}_2$. Then, it computes the *lub* between the types with which the type of $\dot{\tau}_0$ associates the properties in P – $\dot{\tau}$ (provided that such a type exists). The type $\dot{\tau}$ must be a function type. After this, the type system computes the *lub* between the external levels of $\dot{\tau}$, $\dot{\tau}_0$, and $\dot{\tau}_1$ and the level of the program counter, thereby obtaining a level σ. The type system then checks whether σ is lower than or equal to the writing effect declared in the function type $(\dot{\tau})$. This constraint prevents the calling of a function that creates/updates *low* memory depending on *high* values. Finally, the type system checks whether $\dot{\tau}_0$ and $\dot{\tau}_2$ match the type $\dot{\tau}'_0$ of the keyword this and the type $\dot{\tau}'_1$ of the formal parameter, respectively. The whole method call is typed with the return type of the function type $\dot{\tau}'_2$. However, in order to account for possible implicit flows, its external level must be upgraded so that it is higher than or equal to σ.

- [Conditional] In order to type a conditional expression $e_0\ ?\ (e_1) : (e_2)$, the type system first computes the type $\dot{\tau}_0$ of e_0. Then, it types e_1 and e_2 using as the level of the program counter the *lub* between its current level (σ_{pc}) and the external level of $\dot{\tau}_0$. By raising the level of the program counter when typing e_1 and e_2, the type system prevents the creation/update of public resources inside of a branch that was taken depending on secret information. The type given to the whole expression is the *lub* between the types of e_1 and e_2.

- [Sequence] In order to type a sequence expression, the type system first types its two subexpressions. The type given to the whole expression is the type of the second subexpression, since the whole expression evaluates to the value of its second subexpression.

- [Function Literal] A function literal is typed with the type that annotates it. In order to type a function literal, the type system first computes the *lub* between the external level of its type and the current level of the program counter. Then, it checks whether this level is lower than or equal to the writing effect declared in its type. This constraint prevents the evaluation of function literals whose execution updates/creates *low* memory, depending on secret information. Finally, the type system types the body of the function literal using the typing environment obtained by extending the current typing environment with the type $\dot{\tau}'_1$ of the formal argument, the type $\dot{\tau}'_0$ of the keyword this and the types $\dot{\tau}_1$, ..., $\dot{\tau}_n$ of the variables declared in the body of the function literal.

Table 5.3 presents a typing environment and two pairs of programs. While the first pair is type checked by the static type system given in Figure 5.3, the second pair is rejected.

5.3.1 Soundness of the Static Type System

The following lemma states that the execution of a typable expression preserves the well-typing predicate for memories. In other words, the execution of a typable expression in a well-typed memory generates a well-typed memory.

Lemma 5.1 (Well-Typing Preservation)**.** *For any two memories μ and μ', type-based labellings Σ and Σ', reference r, expression e, value v, typing environment Γ, security level σ_{pc}, and security type $\dot{\tau}$, such that:*

- $r \vdash \langle \mu, \Sigma, e\rangle \Downarrow \langle \mu', \Sigma', v\rangle$,
- $\Gamma, \sigma_{pc} \vdash e : \dot{\tau}$,
- *μ is well-typed by Σ and the current scope-chain is well-typed by Γ and Σ;*

It holds that: **(1)** *μ' is well-typed by Σ',* **(2)** *the current scope-chain after the execution of e is well-typed by Γ and Σ', and* **(3)** *if $v \in$ `Ref`, then $\Sigma'(v) \preceq \dot{\tau}$.*

Lemma 5.2 states that the execution of an expression typable using a *high* context level does neither create nor update *low* resources. Hence, the low-projection of the memory that results from the execution of that expression coincides with the low-projection of the initial memory. Finally, the soundness of the proposed type system is established in Theorem 5.1.

Lemma 5.2 (Confinement)**.** *For any two memories μ and μ', type-based labellings Σ and Σ', reference r, expression e, value v, typing environment Γ, security levels σ and σ_{pc}, and security type $\dot{\tau}$, such that:*

- $r \vdash \langle \mu, \Sigma, e\rangle \Downarrow \langle \mu', \Sigma', v\rangle$,
- $\Gamma, \sigma_{pc} \vdash e : \dot{\tau}$,
- *μ is well-typed by Σ,*
- $\sigma_{pc} \not\sqsubseteq \sigma$

It holds that: $\mu \restriction^{\Sigma,\sigma} = \mu' \restriction^{\Sigma',\sigma}$ and $(\mu, r) \restriction^{\Gamma,\sigma} = (\mu', r) \restriction^{\Gamma,\sigma}$.

Theorem 5.1 (Noninterference - Static Type System)**.** *For any expression e and typing environment Γ such that $\Gamma, \sigma_{pc} \vdash e : \dot{\tau}$, it holds that:* $\mathbf{NI}(e, \Gamma)$.

Proofs are given in **Appendix B.1**.

$\mu, r \vDash v \in V$	$\Leftrightarrow$	$v \in V$	% Constant Basic Assertion
$\mu, r \vDash \$v_i \in V$	$\Leftrightarrow$	$r' = \mathsf{Scope}(\mu, r, \$v_i) \ \wedge \ \mu(r' \cdot \mathsf{string}(\$v_i)) \in V$	% Variable Basic Assertion
$\mu, r \vDash \omega_0 \vee \omega_1$	$\Leftrightarrow$	$\mu, r \vDash \omega_0 \ \vee \ \mu, r \vDash \omega_1$	% Disjunction
$\mu, r \vDash \omega_0 \wedge \omega_1$	$\Leftrightarrow$	$\mu, r \vDash \omega_0 \ \wedge \ \mu, r \vDash \omega_1$	% Conjunction
$\mu, r \vDash \neg\omega$	$\Leftrightarrow$	$\mu, r \nvDash \omega$	% Negation
$\mu, r \vDash \mathsf{true}$	$\Leftrightarrow$	*always*	% *Constant True*

Table 5.4: Satisfaction Relation for Runtime Assertions

5.4 Hybrid Information Flow Control in Core JavaScript

The precision of the purely static type system heavily depends on the precision of property set annotations. For instance, a property look-up is typable only if all properties in the corresponding property set annotation are associated with the same raw type. In this section, we modify this type system so as to make its precision independent of the precision of property set annotations. The key insight is that, since our goal is to verify **termination insensitive** noninterference, we can defer failure to execution time. Hence, instead of rejecting a program based on imprecise property set annotations, the hybrid type system infers a set of assertions under which a program can be securely accepted and instruments it so as to dynamically check whether these assertions hold. The instrumented version diverges if the assertions under which the original version was *conditionally accepted* fail to hold at runtime.

5.4.1 A Program Logic for Reasoning about Local Scope

In order to be able to reason about intermediate states of the execution, the type system makes use of an indexed set of variables $\mathtt{V}_{ts} = \{\$v_i\}_{i \in \mathbf{N}}$. These variables are used for bookkeeping the values of intermediate expressions and are not available for the programmer. Since one can easily instrument a program so that it diverges when trying to read/write reserved variables, we can assume that program variables do not overlap with those in $\mathtt{V}_{ts}$. The runtime assertions generated by the type system are described by the following grammar:

$$\omega ::= \$v_i \in V \mid v \in V \mid \mathsf{true} \mid \omega \vee \omega \mid \omega \wedge \omega \mid \neg\omega \tag{5.10}$$

where $\$v_i$ is the i-th variable of $\mathtt{V}_{ts}$ and $V \subset \mathtt{Prim}$ is an arbitrary set of primitive values. The semantics of this logic is given by the satisfaction relation $\vDash$. Informally, $\mu, r \vDash \omega$ means that the assertion ω holds in the memory μ in the scope-chain that starts with the object pointed to by r. The satisfaction relation for assertions is formally given in Table 5.4. We distinguish two types of elementary runtime assertions:

- the *constant basic assertion* $\mu, r \vDash v \in V$ holds provided that $v \in V$,
- the *variable basic assertion* $\mu, r \vDash \$v_i \in V$ holds provided that the value bound to $\$v_i$ in the scope-chain that starts with the object $\mu(r)$ is contained in V.

The remaining assertions are interpreted as in classical propositional logic.

5.4.2 Type Sets and Level Sets

In this section, we use as a running example the program $\mathtt{x}[\mathtt{y}^i] = \mathtt{u}[\mathtt{v}^j] +^k \mathtt{z}$, to be typed using the following typing environment:

$$\begin{aligned} \Gamma(\mathtt{x}) = \dot{\tau}_x &= \mu\kappa.\langle p_0^L : \mathsf{PRIM}^H, p_1^L : \mathsf{PRIM}^L, *^L : \mathsf{PRIM}^L \rangle^L \\ \Gamma(\mathtt{u}) = \dot{\tau}_u &= \mu\kappa.\langle q_0^L : \mathsf{PRIM}^L, q_1^L : \mathsf{PRIM}^H, *^L : \mathsf{PRIM}^H \rangle^L \\ \Gamma(\mathtt{z}) = \Gamma(\mathtt{y}) &= \Gamma(\mathtt{v}) = \mathsf{PRIM}^L \end{aligned} \tag{5.11}$$

This program is not typable using the static type system, because the left-hand side expression is typed with PRIM^L (since the type system uses $\Rsh_\downarrow$ to determine its type), while the right-hand side expression is typed with PRIM^H (since the type system uses $\Rsh_\uparrow$ to determine its type).[2] However, since the property set annotations of this program are very imprecise, it can be the case that the potential illegal flows, which cause the static type system to reject it, never actually happen. Hence, instead of using a single context level when typing a given expression and instead of assigning a single security type to that expression, the hybrid type system uses a set L of *possible* context levels, here called a *level set*, and types each expression with a set T of *possible* security types, here called a *type set*. Each type $\dot{\tau}$ in the type set T and each security level σ in the level set L is paired up with an assertion ω that describes "when" the expression is correctly typed by $\dot{\tau}$ or "when" the context level is in fact σ. For instance, the look-up expressions $\mathtt{x}[\mathtt{y}^i]$ and $\mathtt{u}[\mathtt{v}^j]$ are respectively typed with:

$$T_{\mathtt{x}[\mathtt{y}^i]} = \{(\mathsf{PRIM}^H, \$v_i \in \{p_0\}), (\mathsf{PRIM}^L, \$v_i \in \{p_1\}), (\mathsf{PRIM}^L, \neg(\$v_i \in \{p_0, p_1\}))\} \tag{5.12}$$

$$T_{\mathtt{u}[\mathtt{v}^j]} = \{(\mathsf{PRIM}^L, \$v_j \in \{q_0\}), (\mathsf{PRIM}^H, \$v_j \in \{q_1\}), (\mathsf{PRIM}^H, \neg(\$v_j \in \{q_0, q_1\}))\} \tag{5.13}$$

where $\$v_i$ and $\$v_j$ are the variables of the type system that hold the values to which $\mathtt{y}$ and $\mathtt{v}$ evaluate in their respective context.

It is useful to define a function $\Rsh^?$ that **expects** as input an object type $\dot{\tau}$, a set P of properties to inspect, and an expression e that evaluates to the actual property being inspected[3] and **generates** a set of triples of the form $(\sigma, \dot{\tau}', \omega)$. Each of these triples consists of a security level σ, a security type $\dot{\tau}'$, and the assertion ω that must hold so that the actual property being looked-up has existence level σ and security type $\dot{\tau}'$. Formally, letting $LT^{\dot{\tau},P,e} = \{(\sigma, \dot{\tau}', (e \in \{p\})) \mid p \in P \cap dom(\dot{\tau}) \wedge \Rsh\,(\dot{\tau}, p) = (\sigma, \dot{\tau}')\}$, $\Rsh^?$ is defined as follows:

$$\Rsh^?\,(\dot{\tau}, P, e) = \begin{cases} LT^{\dot{\tau},P,e} & \text{if } P \subseteq dom(\dot{\tau}) \\ LT^{\dot{\tau},P,e} \cup \{(\sigma_*, \tau_*, \neg(e \in dom(\dot{\tau}) \cap P))\} & \text{if } P \not\subseteq dom(\dot{\tau}) \end{cases} \tag{5.14}$$

where $*(\dot{\tau}) = (\sigma_*, \tau_*)$. We extend $\Rsh^?$ to sets of object security types paired up with runtime assertions in the following way:

$$\Rsh^?\,(T, P, e) = \{(\sigma, \dot{\tau}', \omega \wedge \omega') \mid (\dot{\tau}, \omega) \in T \ \wedge\ (\sigma, \dot{\tau}', \omega') \in \Rsh^?\,(\dot{\tau}, P, e)\} \tag{5.15}$$

Given a set LT of triples of the form $(\sigma, \dot{\tau}, \omega)$, we denote by $\pi_{\mathtt{lev}}(LT)$ ($\pi_{\mathtt{type}}(LT)$, resp.) the set of pairs obtained from LT by removing from each triple the security type (the security level, resp.). Observe that $\pi_{\mathtt{type}}(\Rsh^?\,(\dot{\tau}_x, \mathtt{Str}, \$v_i)) = T_{\mathtt{x}[\mathtt{y}^i]}$ and $\pi_{\mathtt{type}}(\Rsh^?\,(\dot{\tau}_u, \mathtt{Str}, \$v_j)) = T_{\mathtt{u}[\mathtt{v}^j]}$.

In the following, we redefine some of the notations previously used with security types and security levels for type sets and level sets. Given a type set T, a level set L, and an assertion ω, we use:

- $lev(T)$ for the level set $\{(\sigma, \omega) \mid (\tau^\sigma, \omega) \in T\}$,
- T^L for the type set $\{(\dot{\tau}', \omega) \mid (\dot{\tau}, \omega_t) \in T \ \wedge\ (\sigma, \omega_l) \in L \ \wedge\ \omega = \omega_t \wedge \omega_l \ \wedge\ \dot{\tau}' = \dot{\tau}^\sigma\}$, and
- T^ω for the type set $\{(\dot{\tau}, \omega \wedge \omega') \mid (\dot{\tau}, \omega') \in T\}$.

5.4.2.1 Combining Type Sets and Level Sets

Since an expression is typed with a set of security types and since the typing rules must consider a set of possible context levels instead of a single context level, the constraints as well as the

[2] Recall that the implicit property set annotation is $\mathtt{Str}$.

[3] Observe that e must either be a variable of the type system or a primitive value.

lub's and *glb's* operations of the static type system must be rewritten in order to account for this change. For instance, in the current running example, the hybrid type system types $\mathtt{u}[\mathtt{v}^j]$ with $T_{\mathtt{u}[\mathtt{v}^j]}$ and z with $T_{\mathtt{z}} = \{(\mathsf{PRIM}^L, \mathsf{true})\}$. Therefore, in order to type $\mathtt{u}[\mathtt{v}]^j +^k \mathtt{z}$, the type system needs to combine two type sets. To this end, we make use of a function $\oplus_{\Cup}$, parameterized with a generic binary function $\Cup$, that given two sets of elements paired up with runtime assertions (be it type sets or level sets), S_0 and S_1, generates a new set $S_0 \oplus_{\Cup} S_1$. Informally, if $(s, \omega) \in S_0 \oplus_{\Cup} S_1$, then, for every memory μ and reference r, $\mu, r \vDash \omega$ if and only if there are two pairs $(s_0, \omega_0) \in S_0$ and $(s_1, \omega_1) \in S_1$ such that $\mu, r \vDash (\omega_0 \wedge \omega_1)$ and $s = s_0 \Cup s_1$. Formally, the operation $\oplus_{\Cup}$ must verify the following:

$$\forall_{\mu \in \mathtt{Mem}, r \in \mathtt{Ref}} \quad \exists_{(s,\omega) \in S_0 \oplus_{\Cup} S_1} \ \mu, r \vDash \omega \ \Leftrightarrow \ \exists_{(s_0,\omega_0) \in S_0, (s_1,\omega_1) \in S_1} \ \mu, r \vDash (\omega_0 \wedge \omega_1) \wedge s = s_0 \Cup s_1 \quad (5.16)$$

Concretely, $T_{\mathtt{u}[\mathtt{v}]^j} \oplus_{\curlyvee} T_{\mathtt{z}} = T_{\mathtt{u}[\mathtt{v}]^j}$. However, making $T'_z = \{(\mathsf{PRIM}^H, \mathsf{true})\}$, it follows that $T_{\mathtt{u}[\mathtt{v}]^j} \oplus_{\curlyvee} T'_z = \{(\mathsf{PRIM}^H, \mathsf{true})\}$.

Constraint Generation In the rules that feature constraints, the hybrid type system tries to infer a dynamic assertion under which the corresponding expression is legal. For instance, when trying to type $\mathtt{x}[\mathtt{y}]^i = \mathtt{u}[\mathtt{v}]^j + \mathtt{z}$, the hybrid type system tries to infer an assertion that is verified only if the level of the property that is being assigned is higher than or equal to the level of the right-hand side expression. To this end, we make use of a function $\mathsf{When}^?_{\Subset}$, parameterized with a generic order relation $\Subset$, that given two sets of elements paired up with runtime assertions, S_0 and S_1, generates an assertion $\omega = \mathsf{When}^?_{\Subset}(S_0, S_1)$. The generated assertion describes the conditions under which there are two pairs $(s_0, \omega_0) \in S_0$ and $(s_1, \omega_1) \in S_1$ such that $s_0 \Subset s_1$ and $\omega_0 \wedge \omega_1$ holds. Formally, if $\omega = \mathsf{When}^?_{\Subset}(S_0, S_1)$, then:

$$\forall_{\mu \in \mathtt{Mem}, r \in \mathtt{Ref}} \ \mu, r \vDash \omega \Leftrightarrow \exists_{(s_0,\omega_0) \in S_0, (s_1,\omega_1) \in S_1} \ \mu, r \vDash (\omega_0 \wedge \omega_1) \ \wedge \ s_0 \Subset s_1 \quad (5.17)$$

For instance, in the current example: $\mathsf{When}^?_{\preceq}(T_{\mathtt{x}[\mathtt{y}^i]}, T_{\mathtt{u}[\mathtt{v}^j]}) = (\$v_i \in \{p_0\}) || (\$v_j \in \{q_0\})$. If $\$v_i \in \{p_0\}$ then the property being assigned is *high* and the assignment is legal. If $\$v_j \in \{q_0\}$, then the value that is being assigned is *low* and, again, the assignment is legal.

5.4.3 Specification of the Type System

The hybrid type system for the imperative fragment of Core JavaScript is presented in Figure 5.4. Typing judgements have the form: $\Gamma, L_{pc} \vdash e \rightsquigarrow {}^{e'}/_{e''} : T$, where:

- Γ is the typing environment,
- L_{pc} a level set that represents all the possible levels of the current context,
- e the expression to be typed,
- e' a new expression semantically equivalent to e except for the executions that are considered illegal,
- e'' an expression that bookkeeps the value to which e' evaluates,
- T the type set representing all possible types of e.

The rules of the hybrid type system have the same structure of the rules of the static type system. While in the static type system the constraints are statically verified, in the dynamic type system, the constraints are statically synthesised and inlined in the program in order to be dynamically verified. However, the constraints are the same.

$$\textsc{Value}\quad \frac{}{\Gamma, L_{pc} \vdash v \rightsquigarrow {}^{v}/_{v} : \{(\mathsf{PRIM}^{\bot}, \mathsf{true})\}} \qquad \textsc{This}\quad \frac{}{\Gamma, L_{pc} \vdash \mathsf{this}^{i} \rightsquigarrow {}^{\$v_i=\mathsf{this}}/_{\$v_i} : \{(\Gamma(\mathsf{this}), \mathsf{true})\}}$$

$$\textsc{Var}\quad \frac{}{\Gamma, L_{pc} \vdash x^{i} \rightsquigarrow {}^{\$v_i=x}/_{\$v_i} : \{(\Gamma(x), \mathsf{true})\}} \qquad \textsc{Object Literal}\quad \frac{\omega = \mathsf{When}^{?}_{\sqsubseteq}(L_{pc}, lev(\dot{\tau})) \qquad e' = \mathsf{Wrap}(\omega, \$v_i = \{\,\}^{\dot{\tau}})}{\Gamma, L_{pc} \vdash \{\,\}^{\dot{\tau},i} \rightsquigarrow {}^{e'}/_{\$v_i} : \{(\dot{\tau}, \mathsf{true})\}}$$

$$\textsc{Binary Operation}\quad \frac{\forall_{i=0,1} \cdot \Gamma, L_{pc} \vdash e_i \rightsquigarrow {}^{e'_i}/_{e''_i} : T_i \qquad e' = e'_0, e'_1, \$v_j = e''_0 \ \mathsf{op}\ e''_1}{\Gamma, L_{pc} \vdash e_0\ \mathsf{op}^{j}\ e_1 \rightsquigarrow {}^{e'}/_{\$v_j} : T_0 \oplus_{\Upsilon} T_1} \qquad \textsc{Variable Assignment}\quad \frac{\Gamma, L_{pc} \vdash e_0 \rightsquigarrow {}^{e'_0}/_{e''_0} : T_0 \qquad \omega = \mathsf{When}^{?}_{\preceq}(T_0^{L_{pc}}, \Gamma(x)) \qquad e = e'_0, \mathsf{Wrap}(\omega, x = e''_0)}{\Gamma, L_{pc} \vdash x = e_0 \rightsquigarrow {}^{e}/_{e''_0} : T_0}$$

$$\textsc{Property Look-up}\quad \frac{\forall_{i=0,1} \cdot \Gamma, L_{pc} \vdash e_i \rightsquigarrow {}^{e'_i}/_{e''_i} : T_i \qquad T_P = \pi_{\mathtt{type}}(\Gamma^{?}\ (T_0, P, e''_1)) \qquad L = lev(T_0) \oplus_{\sqcup} lev(T_1) \qquad e = e'_0, e'_1, \$v_j = e''_0[e''_1]}{\Gamma, L_{pc} \vdash e_0[e_1, P]^{j} \rightsquigarrow {}^{e}/_{\$v_j} : T_P^{L}}$$

$$\textsc{Membership Testing}\quad \frac{\forall_{i=0,1} \cdot \Gamma, L_{pc} \vdash e_i \rightsquigarrow {}^{e'_i}/_{e''_i} : T_i \qquad L = \pi_{\mathtt{lev}}(\Gamma^{?}\ (T_0, P, e''_0)) \oplus_{\sqcup} lev(T_0) \oplus_{\sqcup} lev(T_1) \qquad e = e'_0, e'_1, \$v_j = e''_0\ \mathsf{in}\ e''_1}{\Gamma, L_{pc} \vdash e_0\ \mathsf{in}^{P}_{j}\ e_1 \rightsquigarrow {}^{e}/_{\$v_j} : \mathsf{PRIM}^{L}}$$

$$\textsc{Property Assignment}\quad \frac{\begin{array}{c}\forall_{i=0,1,2} \cdot \Gamma, L_{pc} \vdash e_i \rightsquigarrow {}^{e'_i}/_{e''_i} : T_i \\ LT_P = \Gamma^{?}\ (T_0, P, e''_1) \qquad L_P = \pi_{\mathtt{lev}}(LT) \qquad T_P = \pi_{\mathtt{type}}(LT) \\ \omega_0 = \mathsf{When}^{?}_{\preceq}(T_2, T_P) \qquad \omega_1 = \mathsf{When}^{?}_{\sqsubseteq}(L_{pc} \oplus_{\sqcup} lev(T_0) \oplus_{\sqcup} lev(T_1), L_P) \\ e = e'_0, e'_1, e'_2, \mathsf{Wrap}(\omega_0 \wedge \omega_1, e''_0[e''_1] = e''_2)\end{array}}{\Gamma, L_{pc} \vdash e_0[e_1, P] = e_2 \rightsquigarrow {}^{e}/_{e''_2} : T_2}$$

$$\textsc{Property Deletion}\quad \frac{\begin{array}{c}\forall_{i=0,1} \cdot \Gamma, L_{pc} \vdash e_i \rightsquigarrow {}^{e'_i}/_{e''_i} : T_i \qquad L_P = \pi_{\mathtt{lev}}(\Gamma^{?}\ (T_0, P, e''_0)) \\ \omega = \mathsf{When}^{?}_{\sqsubseteq}(L_{pc} \oplus_{\sqcup} lev(T_0) \oplus_{\sqcup} lev(T_1), L_P) \\ e = e'_0, e'_1, \mathsf{Wrap}(\omega, \$v_i = \mathsf{delete}\ e''_0[e''_1])\end{array}}{\Gamma, L_{pc} \vdash \mathsf{delete}^{i,P}\ e_0[e_1] \rightsquigarrow {}^{e}/_{\mathsf{undefined}} : \mathsf{PRIM}^{\bot}} \qquad \textsc{Sequence}\quad \frac{\forall_{i=0,1} \cdot \Gamma, L_{pc} \vdash e_i \rightsquigarrow {}^{e'_i}/_{e''_i} : T_i \qquad e = e'_0, e'_1}{\Gamma, L_{pc} \vdash e_0, e_1 \rightsquigarrow {}^{e}/_{e''_1} : T_1}$$

$$\textsc{Conditional Expression}\quad \frac{\begin{array}{c}\Gamma, L_{pc} \vdash e_0 \rightsquigarrow {}^{e'_0}/_{e''_0} : T_0 \qquad L'_{pc} = L_{pc} \oplus_{\sqcup} lev(T_0) \\ \forall_{i=1,2} \cdot \Gamma, L'_{pc} \vdash e_i \rightsquigarrow {}^{e'_i}/_{e''_i} : T_i \qquad \omega_{\mathsf{true}} = \neg(e''_0 \in V_F) \qquad \omega_{\mathsf{false}} = (e''_0 \in V_F) \\ T = T_1^{\omega_{\mathsf{true}}} \cup T_2^{\omega_{\mathsf{false}}} \qquad e = e'_0,\ e''_0\ ?\ (e'_1, \$v_j = e''_1) : (e'_2, \$v_j = e''_2)\end{array}}{\Gamma, L_{pc} \vdash e_0\ ?^{j}\ (e_1) : (e_2) \rightsquigarrow {}^{e}/_{\$v_j} : T^{lev(T_0)}}$$

Figure 5.4: Hybrid Typing Secure Information Flow in Core JavaScript

In order to illustrate the difference in functioning between the static and the hybrid type systems, let us consider the Rule [Property Assignment]. In the typing of a property assignment, all of the three subexpressions e_0, e_1, and e_2 are typed with three type sets T_0, T_1, and T_2. The runtime assertion ω_0 checks whether the type of the value being assigned is a subtype of the property of the object to which it is being assigned (thereby avoiding explicit flows). The runtime assertion ω_1 checks whether the existence level of the property being assigned is higher than or equal to the levels of the resources on which the computation of e_0 and e_1 depends (thereby avoiding implicit flows). The instrumentation wraps the property assignment in a conditional expression that checks whether the assertions ω_0 and ω_1 hold.

Original Program	Instrumentation
$\texttt{l1} = \texttt{l2}^{j}\ \mathsf{in}^{\texttt{Str}}\ \texttt{o2}$	$\$v_j = \texttt{l2}$, $(\$v_j\ !== \texttt{"q2"})\ ?$ $(\texttt{l1} = \$v_j\ \mathsf{in}\ \texttt{o2})$ $: (\texttt{\$diverge}())$
$\texttt{o1}[\texttt{l2}^{i}, \{\texttt{"p1"}, \texttt{"p3"}\}] = \texttt{o2}[\texttt{l1}^{j}, \{\texttt{"q2"}, \texttt{"q3"}\}]$	$\$v_i = \texttt{l2}$, $\$v_j = \texttt{l1}$, $(\$v_j === \texttt{"p1"})\ ?$ $(\texttt{o1}[\$v_i] = \texttt{o2}[\$v_j])$ $: (\texttt{\$diverge}())$

Table 5.5: Examples of Programs Accepted by the Hybrid Type System but Rejected by the Static Type System

Inlining Constraints The hybrid type system rewrites the program to be typed in order to dynamically check the assertions under which it is conditionally accepted. To this end, every conditionally typed expression is wrapped in a conditional expression that checks whether the assertion under which it was accepted holds. In order to simplify the specification, we make use of a syntactic function Wrap that given an assertion ω, different from true, and an expression e generates the expression $\omega\ ?\ (e) : (\texttt{\$diverge}())$, where $\texttt{\$diverge}()$ is a runtime function that always diverges. For instance, the program used as the running example is rewritten as follows:

$$\begin{array}{l} \$v_i = \texttt{y}, \$v_j = \texttt{v}, \\ (\$v_i == p_0\ ||\ \$v_j == q_0)\ ?\ \big(\texttt{x}[\$v_i] = \texttt{u}[\$v_j] + \texttt{z}\big) : \big(\texttt{\$diverge}()\big) \end{array} \tag{5.18}$$

If the type system is able to determine that a given constraint is always verified, it generates the assertion true. In that case, Wrap simply outputs the given expression.

Table 5.5 shows that the hybrid type system type checks the two programs of Table 5.3 that the static type system does not type check.

5.4.3.1 Soundness of the Hybrid Type System

In order to prove the correctness of the type system, one must be able to relate the memory that results from the execution of a program and the memory that results from the execution of its instrumented version (generated by the hybrid type system). To this end, we introduce a *similarity* relation between the memories obtained from the execution of original programs and the memories obtained from the execution of their instrumented counterparts. Informally, two memories μ and μ' are said to be *hts-similar*, written $\mu \simeq_{hts} \mu'$, if μ does not bind type system variables (in $\mathtt{V}_{ts}$) and the two memories only differ in $\mathtt{V}_{ts}$.

Definition 5.7 (hts-Similarity)**.** *A memory μ is said to be hts-similar to a memory μ', written $\mu \simeq_{hts} \mu'$, if and only if $dom(\mu) = dom(\mu')$ and for every reference $r \in dom(\mu)$, it holds that:*

- *If $\mu(r)$ is a scope object: $\forall_{p \in dom(\mu(r))}\ \mathsf{ident}(p) \notin \mathtt{V}_{ts}$,*
- *If $\mu'(r)$ is* **not** *a scope object: $\forall_{p \in dom(\mu'(r))}\ \mu(r \cdot p) = \mu'(r \cdot p)$,*

- *If* $\mu'(r)$ *is a scope object:* $\forall_{p \in dom(\mu'(r))}\ \mathsf{ident}(p) \notin \mathcal{V}_{ts} \Rightarrow \mu(r \cdot p) = \mu'(r \cdot p)$.

The soundness of the hybrid type system is established by Theorems 5.2 and 5.3. The former states that the expressions generated by the type system preserve the semantics of their original counterparts (in other words, the semantics of the instrumented program is contained in the semantics of the original one), while the latter states that instrumented program is noninterferent.

Theorem 5.2 (Transparency)**.** *For any expression* e*, typing environment* Γ*, memory* μ *well-typed by* Σ*, level set* L*, and reference* r *such that:*

- $\Gamma, L \vdash e \rightsquigarrow {}^{e'}\!/_{e''} : T$ *and*
- $r \vdash \langle \mu, \Sigma, e' \rangle \Downarrow \langle \mu'_f, \Sigma_f, v \rangle$*;*

It holds that there exists a memory μ_f *such that* $r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu_f, \Sigma_f, v \rangle$ *and* $\mu_f \simeq_{hts} \mu'_f$.

Theorem 5.3 (Noninterference)**.** *For any expression* e*, typing environment* Γ*, and level set* L*, if* $\Gamma, L \vdash e \rightsquigarrow {}^{e'}\!/_{e''} : T$*, then* $\mathbf{NI}(e', \Gamma)$.

Proofs are given in **Appendix B.2.**

5.5 Related Work

Static Type Systems for Securing Information Flow Since the seminal work of Volpano et al. [Volpano 1996] on typing secure information flow in a simple imperative language, type systems for information flow control have been proposed for a wide variety of languages, ranging from functional [Almeida Matos 2009, Pottier 2002] to Java-like object-oriented languages [Banerjee 2002]. To the best of our knowledge, our static type system for enforcing secure information flow in Core JavaScript is the first that addresses the particular features of JavaScript in the context of information flow control.

Hybrid Monitors *Hybrid information flow monitors* [Venkatakrishnan 2006, Guernic 2007, Shroff 2007], use static analysis to reason about the implicit flows that arise due to untaken execution paths. In fact, hybrid monitors must either statically or dynamically estimate the resources that are created/updated in untaken program paths. More concretely, after the execution of a control-flow expression (such as a function call, a method call, or a conditional expression) that depends on *high* information, the security levels of the resources that would have been updated in alternative paths must be set to *high*. The dynamic features of JavaScript make it very difficult to design the type of static analysis required by hybrid monitors.

Hybrid monitors can also be used as a means to mitigate the performance overhead imposed by runtime monitoring. For instance, Moore et al. [Moore 2011] showed how to combine monitoring and static analysis so as to reduce the number of resources whose labels are tracked at runtime.

Interestingly, Russo et al. [Russo 2010] proved that hybrid monitors are more permissive than both purely dynamic and purely static enforcement mechanisms. Indeed, their result supports the need for mechanisms which combine static and dynamic analysis like the hybrid type system we present in the chapter. However, unlike hybrid monitors, the hybrid type system we propose does **not** require any kind of runtime tracking of security levels (since the inlined conditions feature the actual values that are computed by the program rather than their levels).

Gradual Typing Secure Information Flow Recently, *gradual security typing* [Disney 2011, Fennell 2013] has been proposed as a way to combine runtime monitoring and static analysis in order to cater for controlled forms of *polymorphism*. Concretely, the programmer is expected to introduce runtime casts in points where values of a pre-determined security type are expected. "The type system statically guarantees adherence to the [security] policies on the static side of a cast, whereas the runtime system checks the policies on the dynamic side" [Fennell 2013]. Like the hybrid type system presented in this chapter, this approach can be used to overcome the problem of property names computed at runtime. However, the use of gradual typing would necessarily imply partial tracking of security levels.

Static Analysis for JavaScript There is plenty of literature on the subject of static analysis for JavaScript. Thiemann [Thiemann 2005] proposed a type system to guarantee *termination* and *progress* for a JavaScript-like language. Indeed, we borrow from [Thiemann 2005] the notion of *default type*. The type system presented in [Thiemann 2005] does not account for objects whose domain may change at runtime. To overcome this issue, Anderson et al. [Anderson 2005] have proposed a type inference algorithm that allows objects "to evolve in a controlled manner" by classifying their properties as *definite* or *potential*. This additional information could be used by the static type system to distinguish *property creations* from *property updates*, thereby relaxing the constraints imposed on property updates, which would not need to take into account the existence level of the updated property.

Later, Jensen et al. [Jensen 2009] presented the first sound and detailed tool for type analysis in real JavaScript code, called TAJS. The proposed type analysis for JavaScript is flow-sensitive and based on abstract interpretation. The main contribution of this analysis are the design of a complex lattice structure that caters for the particular features of the language and the development of a prototype that covers the JavaScript full language.

The TypeScript programming language [Microsoft 2014] adds optional types to JavaScript, with support for interaction with existing JavaScript libraries via interface declarations. The main idea of this language is to harness the flexibility of real JavaScript, while at the same time providing some of the advantages otherwise reserved for statically typed languages, such as informative compiling errors and automatic code completion. Furthermore, types can be also used for documentation purposes. Since client-side JavaScript programs make heavy use of external APIs that are not available for static typing, the analysis of TypeScript programs requires the specification of *interface declarations* for the external libraries that a program may use. These *interface declarations* are, however, written by hand and often not by the authors of the libraries. This is an error-prone process that has severe consequences, since the fact that an interface declaration is incorrectly specified causes the tools that depend on it to produce wrong results (for example, wrong suggestions for autocompletion). To solve this program, Feldthaus et al. [Feldthaus 2014] have recently proposed an analysis for checking the correction of TypeScript declaration files with respect to JavaScript library implementations. One of the contributions of this work is a formalisation of the TypeScript typing language. Interestingly, this language includes a generalisation of *default type* for objects that is called *indexer*. Concretely, an indexer allows for the specification of classes of properties in a same object type that are to be assigned to the same type.

Static Analysis for Securing JavaScript Applications Due to the complexity of JavaScript semantics, most mechanisms for preventing security violations spawned by client-side JavaScript code have focused on isolation properties [Maffeis 2009, Politz 2011], which are easier to enforce than noninterference [Goguen 1982]. The analyses presented in [Maffeis 2009] and [Politz 2011] deal in different ways with the issue of property names computed at runtime. While the authors of [Maffeis 2009] consider a subset of the language that does not include this

kind of look-up expression, the type system presented in [Politz 2011] overapproximates the set of properties to which these arbitrary expressions may evaluate. We believe that the idea illustrated by the hybrid type could be applied both to [Maffeis 2009] and [Politz 2011] in order to increase their permissiveness.

Keil et al. [Keil 2013] presented a type-based flow-sensitive dependency analysis for securing information flow based on TAJS [Jensen 2009]. This analysis is intended to be used for security purposes. The authors formalise the analysis as "an abstraction of a tainting semantics" and prove the soundness of the abstraction, a non-interference property, and the termination of the analysis.

Chapter 6

An Extensible Monitored Semantics for Securing Web APIs

Contents

Although JavaScript can be used as a general-purpose programming language, many JavaScript programs are designed to be executed in a browser in the context of a web page. Such programs often interact with the web page in which they are included, as well as the browser itself, through Application Programming Interfaces (APIs). Some APIs are fully implemented in JavaScript, whereas others are built with a mix of different technologies, which can be exploited to conceal sophisticated security violations. Thus, understanding the behaviour of client-side web applications as well as proving their compliance with a given security policy requires cross-language reasoning that is often far from trivial.

The size, complexity, and number of commonly used APIs poses an important challenge to any attempt at formally reasoning about the security of JavaScript programs [Guha 2012]. To tackle this problem, we propose a methodology for extending JavaScript monitored semantics. This methodology allows us to verify whether a monitor complies with the proposed noninterference property in a modular way. Thus, we make it possible to prove that a security monitor is still noninterferent when extending it with a new API, without having to revisit the whole model.

Generally, an API can be viewed as a particular set of specifications that a program can follow to make use of the resources provided by another particular application. For client-side JavaScript programs, this definition of API applies both to:

- interfaces of services that are provided to the program by the environment in which it executes, namely the web browser (for instance, the DOM, the XMLHttpRequest, and the W3C Geolocation APIs);
- interfaces of JavaScript libraries that are explicitly included by the programmer (for instance, jQuery, Prototype.js, and Google Maps Image API).

In the context of this work, the main difference between these two types of APIs is that in the former case their semantics escapes the JavaScript semantics, whereas in the latter it does not.

The methodology proposed here was designed as a generic way of extending security monitors to deal with the first type of APIs. Nevertheless, it can also be applied to the second of APIs in order to avoid the monitoring of the library's code.

Outline This chapter is structured as follows: Section 6.1 introduces an extensible Core JavaScript semantics based on the semantics introduced in Chapter 2. The extensible semantics is formalised in a way that makes it possible to extend it effortlessly with arbitrary external APIs. Section 6.2 presents an extensible version of the monitored semantics described in Chapter 4. Furthermore, this section describes the criteria that a monitored API needs to verify so that the plugging of the this API into the extensible monitor yields a noninterferent monitor. Section 7.4 discusses related work. Finally, in Section 7.5, we analyse the following questions: **(1)** How can one inline the extensible monitor presented in this chapter? **(2)** How general is our definition of confinement for API plugins? The second question is of particular interest because, as we shall see, the definition of confinement for APIs depends on the features of the proposed mechanism for plugging APIs into the language runtime.

6.1 An Extensible Semantics for Core JavaScript

At the formal level, in order to model the execution of APIs that may escape the JavaScript semantics, we extend the semantics of Core JavaScript $\Downarrow$ with two alternative rules for property look-ups and method calls, thereby obtaining a new big-step semantic relation $\Downarrow^{\mathsf{API}}$. The alternative rules cater for the execution of arbitrary external APIs. Concretely, upon the invocation of a method, the new semantics checks whether it is a standard method or rather a method belonging to an API. In the former case, the semantics proceeds as before, whereas in the latter it uses the semantics of that particular API to compute its return value. Analogously, when looking-up the value of an object's property, the semantics checks whether that property look-up should be handled by an external API (rather than the JavaScript engine) in which case it uses the semantics of that particular API to compute the value yielded by that property look-up.

Formally, we define an API $\mathsf{API} \in \mathcal{A}$ as a triple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R} \rangle$ consisting of:

- a set $\mathcal{S}$ of API states that model the state of the API,
- a set $\mathcal{P}$ of *API plugins* that model the behaviours of the API,
- a mapping $\mathcal{R}$, that we call API register, used to determine when to apply each API plugin.

Concretely, an API register $\mathcal{R} : \texttt{Ref} \times \texttt{Prim} \to \mathcal{P}$ is a partial function that maps a pair consisting of a reference and a primitive value to an API plugin. In the following, we assume that expressions are marked with arbitrary annotations taken from a set `Ann`. These annotations are used by the programmer to provide additional information to the API plugins. Given an API $\mathsf{API} = \langle \mathcal{S}, \mathcal{P}, \mathcal{R} \rangle$, we use $\mathsf{API.Reg}$ to refer to $\mathcal{R}$.

An API plugin can be seen as a function that, given a sequence of values, updates the current API state and produces a new value. Formally, we model an API plugin $\mathsf{pg} \in \mathcal{P}$ as a relation of the form:

$$\langle \nu, \overrightarrow{v} \rangle^{\alpha} \ \mathsf{pg} \ \langle \nu', v \rangle^{\beta} \tag{6.1}$$

where:

- $\nu, \nu' \in \mathcal{S}$ are the API states immediately before and after the execution of pg,
- $\overrightarrow{v}$ is the sequence of *arguments* given to pg,

$\alpha \in \texttt{Ann}$	% Syntactic Annotations
$\beta \in \texttt{Ev}$	% Internal Events
$\nu \in \mathcal{S}$	% API States
$\texttt{pg} \in \mathcal{P} \subseteq (\mathcal{S} \times \texttt{Val}^* \times \texttt{Ann}) \times (\mathcal{S} \times \texttt{Val} \times \texttt{Ann})$	% API Plugins
$\mathcal{R} : \texttt{Ref} \times \texttt{Prim} \to \mathcal{P}$	% API Register
$\mathsf{API} = \langle \mathcal{S}, \mathcal{P}, \mathcal{R} \rangle \in \mathcal{A}$	% API
$\Downarrow^{\mathsf{API}}$	% Extended Core JavaScript Semantics

Table 6.1: Components of the API Formal Model

- $\alpha \in \texttt{Ann}$ is the syntactic annotation of the expression that triggered the call to pg,
- v is the *return value* of the call to pg, and
- $\beta \in \texttt{Ev}$ is an internal event used by the security monitor and explained in detail Section 6.2.

The components of the API formal model are systematised in Table 6.1.

The API register plays a key role in this model, since it is its job to **plug the API plugins into the extensible semantics**. Indeed, the API register identifies the property look-ups and method calls that trigger the execution of an API plugin. In such cases, it also the job of the API register to determine which API plugin is to be executed.

But how can the API register differentiate the property look-ups/method calls that trigger the execution of an API plugin from the ones that do not? In order to make this possible, we assume that the internal "objects" that compose an API state ν (and which do not have to be Core JavaScript objects) are allocated in a set of references that does not overlap with the set of references used for the allocation of Core JavaScript objects. However, these references can be returned by any API plugin. Hence, Core JavaScript expressions may refer to the internal "objects" of a given API state via API references. For an expression to trigger the invocation of an API plugin, it suffices that it interacts with a reference that belongs to the corresponding API (in a pre-established way). Consequently, when extending Core JavaScript with an API, the initial Core JavaScript memory is assumed to contain at least one API reference that serves as an entry point to the whole API.

The extended semantics intercepts property look-ups and methods calls. It then uses **the values of the first two subexpressions** of the intercepted expression to determine whether the evaluation of that expression triggers the execution of an API plugin and, if so, which API plugin to apply. Concretely, in order to check whether the evaluation of a given expression triggers the execution of an API plugin, the semantics checks whether the pair consisting of the values of its first two subexpressions is in the domain of the API register. If that is the case, the extensible semantics applies the API register to those two values, thereby obtaining the API plugin to execute.

The semantics of Core JavaScript extended with an arbitrary API $\mathsf{API} = \langle \mathcal{S}, \mathcal{P}, \mathcal{R} \rangle \in \mathcal{A}$ is presented in Figure 6.1. Since the semantics must take into account the API state, which can change during the execution, initial and final configurations are extended with an API state. Concretely, the transitions of the extended monitor have the following form:

$$r \vdash \langle \mu, e \mid \nu \rangle \Downarrow^{\mathsf{API}} \langle \mu', v \mid \nu' \rangle \tag{6.2}$$

where: ν and ν' are the initial and final API states. The remaining elements keep their original meanings. Observe that the API register does not change during the execution.

$$\textsc{Property Look-up}$$
$$\frac{r \vdash \langle \mu, e_0 \mid \nu \rangle \Downarrow^{\mathsf{API}} \langle \mu_0, r_0 \mid \nu_0 \rangle \qquad r \vdash \langle \mu_0, e_1 \mid \nu_0 \rangle \Downarrow^{\mathsf{API}} \langle \mu_1, m_1 \mid \nu_1 \rangle \quad \langle r_0, m_1 \rangle \notin dom(\mathsf{API.Reg}) \qquad r' = \mathsf{Proto}(\mu_1, r_0, m_1) \quad r' \neq \mathsf{null} \Rightarrow v = \mu_1(r')(m_1) \qquad r' = \mathsf{null} \Rightarrow v = \mathsf{undefined}}{r \vdash \langle \mu, e_0[e_1]^\alpha \mid \nu \rangle \Downarrow^{\mathsf{API}} \langle \mu_1, v \mid \nu_1 \rangle}$$

$$\textsc{External Property Look-up}$$
$$\frac{r \vdash \langle \mu, e_0 \mid \nu \rangle \Downarrow^{\mathsf{API}} \langle \mu_0, r_0 \mid \nu_0 \rangle \qquad r \vdash \langle \mu_0, e_1 \mid \nu_0 \rangle \Downarrow^{\mathsf{API}} \langle \mu_1, v_1 \mid \nu_1 \rangle \quad \langle r_0, v_1 \rangle \in dom(\mathsf{API.Reg}) \qquad \mathsf{pg} = \mathsf{API.Reg}(r_0, v_1) \qquad \langle \nu_1, r_0 :: v_1 \rangle^\alpha \ \mathsf{pg} \ \langle \nu', v \rangle^\beta}{r \vdash \langle \mu, e_0[e_1]^\alpha \mid \nu \rangle \Downarrow^{\mathsf{API}} \langle \mu_1, v \mid \nu' \rangle}$$

$$\textsc{Method Call}$$
$$\frac{r \vdash \langle \mu, e_0 \mid \nu \rangle \Downarrow^{\mathsf{API}} \langle \mu_0, r_0 \mid \nu_0 \rangle \qquad r \vdash \langle \mu_0, e_1 \mid \nu_0 \rangle \Downarrow^{\mathsf{API}} \langle \mu_1, m_1 \mid \nu_1 \rangle \quad r \vdash \langle \mu_1, e_2 \mid \nu_1 \rangle \Downarrow^{\mathsf{API}} \langle \mu_2, v_2 \mid \nu_2 \rangle \qquad \langle r_0, m_1 \rangle \notin dom(\mathsf{API.Reg}) \qquad r_m = \mathsf{Proto}(\mu_2, r_0, m_1) \quad r_f = \mu_2(r_m \cdot m_1) \qquad \langle \hat{\mu}, \hat{e}, \hat{r} \rangle = \mathsf{NewScope}(\mu_2, r_f, v_2, r_0) \qquad \hat{r} \vdash \langle \hat{\mu}, \hat{e} \mid \nu_2 \rangle \Downarrow^{\mathsf{API}} \langle \mu', v \mid \nu' \rangle}{r \vdash \langle \mu, e_0[e_1](e_2)^\alpha \mid \nu \rangle \Downarrow^{\mathsf{API}} \langle \mu', v \mid \nu' \rangle}$$

$$\textsc{External Method Call}$$
$$\frac{r \vdash \langle \mu, e_0 \mid \nu \rangle \Downarrow^{\mathsf{API}} \langle \mu_0, r_0 \mid \nu_0 \rangle \qquad r \vdash \langle \mu_0, e_1 \mid \nu_0 \rangle \Downarrow^{\mathsf{API}} \langle \mu_1, v_1 \mid \nu_1 \rangle \quad r \vdash \langle \mu_1, e_2 \mid \nu_1 \rangle \Downarrow^{\mathsf{API}} \langle \mu_2, v_2 \mid \nu_2 \rangle \qquad \langle r_0, m_1 \rangle \in dom(\mathsf{API.Reg}) \quad \mathsf{pg} = \mathsf{API.Reg}(r_0, v_1) \qquad \langle \nu_2, r_0 :: v_1 :: v_2 \rangle^\alpha \ \mathsf{pg} \ \langle \nu', v \rangle^\beta}{r \vdash \langle \mu, e_0[e_1](e_2)^\alpha \mid \nu \rangle \Downarrow^{\mathsf{API}} \langle \mu', v \mid \nu' \rangle}$$

Figure 6.1: An Extensible Semantics for Core JavaScript

Since the API register only intercepts property look-ups and method calls, only their corresponding rules may have a different semantics from the one presented in Chapter 2. These rules are presented in Figure 6.1 and briefly described below.

- In the Rules [Property Look-up] and [External Property Look-up], the semantics starts by sequentially evaluating the two subexpressions of the current expression, thereby obtaining: **(1)** the reference r_0 of the object whose property is being inspected and **(2)** a value v_1 (which, in the case of a standard property look-up, corresponds to that property's name). The semantics then checks whether (r_0, v_1) is in the domain of the API register $\mathsf{API.Reg}$. If that is the case, the corresponding API plugin $\mathsf{pg} = \mathsf{API.Reg}(r_0, v_1)$ is applied and the whole expression evaluates to its return value. Otherwise, the semantics proceeds as in the semantics of Core JavaScript (described in Chapter 2).
- In the Rules [Method Call] and [External Method Call], the semantics starts by sequentially evaluating the three subexpressions of the current expression, thereby obtaining: **(1)** the reference r_0 of the object on which the method is called, **(2)** a value v_1 (which, in the case of a standard property method-call, corresponds to the name of the method), and **(3)** the value v_2 to be used as the argument. The semantics then checks whether (r_0, v_1) is in the domain of the API register $\mathsf{API.Reg}$. If that is the case, the corresponding API plugin $\mathsf{pg} = \mathsf{API.Reg}(r_0, v_1)$ is applied and the whole expression evaluates to its return value. Otherwise, the semantics proceeds as in the semantics of Core JavaScript (described in Chapter 2).

6.2 A Secure Extensible Monitor for Core JavaScript

Having shown how to extend the Core JavaScript semantics in order to take into account the execution of APIs that may take place outside the perimeter of the JavaScript engine, we now show how to extend its monitored version presented in Chapter 4.

In order to extend the JavaScript monitored semantics presented in Chapter 4, each API state ν is paired up with an abstract state $\Xi \in \mathcal{S}_{lab}$, that we call API labelling, which labels the resources of ν with security levels. Hence, given an API state ν paired up with an API labelling Ξ, the API labelling Ξ establishes, for each security level σ, the part of that API state that an attacker at level σ can see. Different APIs have different types of resources and therefore label those resources in different ways. Hence, we do not concretise the set of API labellings $\mathcal{S}_{lab}$.

In order to plug arbitrary APIs into the monitored semantics, we propose to associate each API plugin $\mathsf{pg} \in \mathcal{P}_{lab}$ with a monitoring counterpart $\mathsf{pg}_{lab} \in \mathcal{P}_{lab}$, that we call *API monitor plugin*. An API monitor plugin pg_{lab} establishes how the API labelling Ξ should be updated after the execution of its corresponding API plugin pg. Informally, while the API plugin pg updates the API state, its monitoring counterpart pg_{lab} updates the API labelling. A pair $(\mathsf{pg}, \mathsf{pg}_{lab}) \in \mathcal{P} \times \mathcal{P}_{lab}$, consisting of an API plugin and a monitor API plugin is called a *monitored API plugin*. The API monitor plugin pg_{lab} uses the internal event $\beta \in \mathtt{Ev}$ generated by its corresponding API plugin pg as well as the security levels of the arguments given to pg in order to determine how the API labelling should be updated. Hence, an API monitor plugin is modelled as a relation of the form:

$$\langle \Xi, \overrightarrow{\sigma} \rangle^{\beta} \ \mathsf{pg}_{lab} \ \langle \Xi', \sigma \rangle \tag{6.3}$$

where:

- Ξ, Ξ' are the API labellings immediately before and after the execution of the API plugin,
- $\overrightarrow{\sigma}$ is the sequence of levels of the arguments given to the API plugin,

$\Xi \in \mathcal{S}_{lab}$	% API Labellings
$\mathsf{pg}_{lab} \in \mathcal{P}_{lab}$	% API Monitor Plugins
$(\mathsf{pg}, \mathsf{pg}_{lab}) \in \mathcal{P} \times \mathcal{P}_{lab}$	% Monitored API Plugins
$\mathcal{R}_{IF} : \mathtt{Ref} \times \mathtt{Prim} \to \mathcal{P} \times \mathcal{P}_{lab}$	% Monitored API Register
$\sim_{api} \in (\mathcal{S} \times \mathcal{S}_{lab}) \times \mathcal{L} \times (\mathcal{S} \times \mathcal{S}_{lab})$	% API Low-Equality Relation
$\mathsf{API}_{IF} = \langle \mathcal{S}, \mathcal{S}_{lab}, \mathcal{P}, \mathcal{P}_{lab}, \mathcal{R}_{IF}, \sim_{api} \rangle \in \mathcal{A}_{lab}$	% Monitored API
$\Downarrow_{IF}^{\mathsf{API}}$	% Extended Monitored Semantics

Table 6.2: Components of the Monitored API Formal Model

- β is the internal event generated by the API plugin in order to provide additional information to its monitoring counterpart, and
- σ is the security level that labels the return value (called the *reading effect* of the API plugin).

In order for an API register to be used by the extensible monitored semantics, it must output both the API plugin and the API monitor plugin. Hence, we define *a monitored API register* as a function $\mathcal{R}_{IF} : \mathtt{Ref} \times \mathtt{Prim} \to \mathcal{P} \times \mathcal{P}_{lab}$ that given a reference and a primitive value outputs a monitored API plugin. Finally, we define a monitored API $\mathsf{API}_{IF} \in \mathcal{A}_{lab}$ as tuple $\langle \mathcal{S}, \mathcal{S}_{lab}, \mathcal{P}, \mathcal{P}_{lab}, \mathcal{R}_{IF}, \sim_{api} \rangle$ consisting of:

- a set $\mathcal{S}$ of API states,
- a set $\mathcal{S}_{lab}$ of API labellings,
- a set $\mathcal{P}$ of *API plugins*,
- a set $\mathcal{P}_{lab}$ of *API monitor plugins*,
- a monitored API register $\mathcal{R}_{IF}$, and
- a low-equality relation $\sim_{api}$ between labelled API states (described in the following section).

We use $\mathsf{API}_{IF}.\mathsf{Reg}$ and $\mathsf{API}_{IF}.\mathsf{equality}$ to denote, respectively, the monitored API register and the low-equality relation $\sim_{api}$ of the API API_{IF}. The components of the monitored API formal model are systematised in Table 6.2.

A configuration of the monitored semantics for extended Core JavaScript is obtained by adding both to the initial and final configurations of the original monitor an API state ν and an API labelling Ξ. The rules of the extended monitored semantics $\Downarrow_{IF}^{\mathsf{API}}$ have the form:

$$r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \mid \nu, \Xi \rangle \Downarrow_{IF}^{\mathsf{API}} \langle \mu', v, \Sigma', \sigma \mid \nu', \Xi' \rangle \tag{6.4}$$

where: ν and ν' are the initial and final API states and Ξ and Ξ' are the initial and final API labellings. The remaining elements keep their original meanings. Figure 6.2 presents the rules of the monitor that applies the API plugins. Since the two rules are very similar, only the Rule [External Property Look-up] is described.

- [External Property Look-up] The monitored semantics starts by sequentially evaluating the two subexpressions of the current expression, thereby obtaining: **(1)** the reference r_0

$$\textsc{External Property Look-up}$$
$$\frac{\begin{array}{c}\forall_{i=0,1}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \mid \nu_i, \Xi_i\rangle \Downarrow_{IF}^{\mathsf{API}_{IF}} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \mid \nu_{i+1}, \Xi_{i+1}\rangle \\ \langle v_0, v_1\rangle \in dom(\mathsf{API}_{IF}.\mathsf{Reg}) \qquad (\mathsf{pg}, \mathsf{pg}_{lab}) = \mathsf{API}_{IF}.\mathsf{Reg}(v_0, v_1) \\ \langle \nu_2, v_0 :: v_1\rangle^{\alpha}\ \mathsf{pg}\ \langle \nu', v\rangle^{\beta} \qquad \langle \Xi_2, \sigma_0 :: \sigma_1\rangle^{\beta}\ \mathsf{pg}_{lab}\ \langle \Xi', \sigma\rangle\end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, e_0[e_1]^{\alpha}, \Sigma_0 \mid \nu_0, \Xi_0\rangle \Downarrow_{IF}^{\mathsf{API}_{IF}} \langle \mu_2, v, \Sigma_2, \sigma \mid \nu', \Xi'\rangle}$$

$$\textsc{External Method Call}$$
$$\frac{\begin{array}{c}\forall_{i=0,1,2}\ r, \sigma_{pc} \vdash \langle \mu_i, e_i, \Sigma_i \mid \nu_i, \Xi_i\rangle \Downarrow_{IF}^{\mathsf{API}_{IF}} \langle \mu_{i+1}, v_i, \Sigma_{i+1}, \sigma_i \mid \nu_{i+1}, \Xi_{i+1}\rangle \\ \langle v_0, v_1\rangle \in dom(\mathsf{API}_{IF}.\mathsf{Reg}) \qquad (\mathsf{pg}, \mathsf{pg}_{lab}) = \mathsf{API}_{IF}.\mathsf{Reg}(v_0, v_1) \\ \langle \nu_3, v_0 :: v_1 :: v_2\rangle^{\alpha}\ \mathsf{pg}\ \langle \nu', v\rangle^{\beta} \qquad \langle \Xi_3, \sigma_0 :: \sigma_1 :: \sigma_2\rangle^{\beta}\ \mathsf{pg}_{lab}\ \langle \Xi', \sigma\rangle\end{array}}{r, \sigma_{pc} \vdash \langle \mu_0, e_0[e_1](e_2)^{\alpha}, \Sigma_0 \mid \nu_0, \Xi_0\rangle \Downarrow_{IF}^{\mathsf{API}_{IF}} \langle \mu_3, v, \Sigma_3, \sigma \mid \nu', \Xi'\rangle}$$

Figure 6.2: An Extensible Monitored Semantics for Core JavaScript

of the object whose property is being inspected labelled by σ_0 and **(2)** a value v_1 (which, in the case of the standard property look-up corresponds to the property's name) labelled by σ_1. The semantics then checks whether (r_0, v_1) is in the domain of the monitored API register $\mathsf{API}_{IF}.\mathsf{Reg}$. If that is the case, the corresponding monitored API plugin $(\mathsf{pg}, \mathsf{pg}_{lab}) = \mathcal{R}_{IF}(r_0, v_1)$ is obtained from the API register. Subsequently, the API plugin $\mathsf{pg} = \mathcal{R}(r_0, m_1)$ is applied, generating a value v and an internal event β. The API monitor plugin is then applied (with the internal event β and the levels σ_0 and σ_1), generating a level σ. The whole expression evaluates to v and has reading effect σ.

6.2.1 An Attacker Model for External APIs?

In order to state the properties of the extensible monitor, one must be able to relate the API states that result from the execution of the same program starting from two low-equal memories. This, however, requires being able to relate the API states that result from the execution of that program. To this end, we assume that every monitored API comes equipped with a low-equality relation $\sim_{api}$ between labelled API states (and parameterizable with a security level σ). Informally, given two API states ν_0 and ν_1 respectively labelled by Ξ_0 and Ξ_1, and a security level σ, $\nu_0, \Xi_0 \sim_{api}^{\sigma} \nu_1, \Xi_1$ means that ν_0 and ν_1 are indistinguishable for an attacker at level σ. The only restriction that we impose on $\sim_{api}$ is that, for every security level $\sigma \in \mathcal{L}$, the relation $\sim_{api}^{\sigma}$ be an **equivalence relation** [Davey 2002].

6.2.2 Noninterference for Monitored APIs

Noninterference for Extended Monitors Definitions 6.1 and 6.2 adapt the notions of monitor confinement and monitor noninterference (given in Chapter 4) to monitors extended with arbitrary APIs. The main difference between the new definitions with respect to those presented in Chapter 4 is that the new definitions take into account the labelled API state. To this end, they make use of an abstract low-equality relation (as discussed in the previous section).

Definition 6.1 (Confined Extended Monitor)**.** *Given a monitored API* $\mathsf{API}_{IF} \in \mathcal{A}_{lab}$*, the extended monitored semantics* $\Downarrow_{IF}^{\mathsf{API}_{IF}}$ *is said to be confined if, for any expression* e*, memory* μ*, labelling* Σ*, API state* ν*, API labelling* Ξ*, reference* r*, and security levels* σ_{pc} *and* σ*, such that:*

- $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \mid \nu, \Xi\rangle \Downarrow_{IF}^{\mathsf{API}} \langle \mu', v', \Sigma', \sigma' \mid \nu', \Xi'\rangle$

- $\sigma_{pc} \not\sqsubseteq \sigma$

It holds that: $\mu_f, \Sigma_f \sim_\sigma \mu', \Sigma'$, $\nu, \Xi \sim^\sigma_{api} \nu'_f, \Xi'_f$, *and* $\sigma' \not\sqsubseteq \sigma$, *where* $\sim_{api} = \mathsf{API}_{IF}.\mathsf{equality}$.

Definition 6.2 (Noninterferent Extended Monitor)**.** *Given a monitored API* $\mathsf{API}_{IF} \in \mathcal{A}_{lab}$, *the extended monitored semantics* $\Downarrow^{\mathsf{API}_{IF}}_{IF}$ *is said to be noninterferent, written* $\mathbf{NI}(\Downarrow^{\mathsf{API}_{IF}}_{IF})$, *if for any expression* e, *memories* μ *and* μ' *respectively labelled by* Σ *and* Σ', *API states* ν *and* ν' *respectively labelled by* Ξ *and* Ξ', *reference* r, *and security levels* σ_{pc} *and* σ, *such that:*

- $\mu, \Sigma \sim_\sigma \mu', \Sigma'$,
- $\nu, \Xi \sim^\sigma_{api} \nu', \Xi'$,
- $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \mid \nu, \Xi\rangle \Downarrow^{\mathsf{API}}_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \mid \nu_f, \Xi_f\rangle$,
- $r, \sigma_{pc} \vdash \langle \mu', e, \Sigma' \mid \nu', \Xi'\rangle \Downarrow^{\mathsf{API}}_{IF} \langle \mu'_f, v'_f, \Sigma'_f, \sigma'_f \mid \nu'_f, \Xi'_f\rangle$

It holds that: $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$, $\nu_f, \Xi_f \sim^\sigma_{api} \nu'_f, \Xi'_f$, *and* $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$, *where* $\sim_{api} = \mathsf{API}_{IF}.\mathsf{equality}$.

Noninterference for Monitored APIs In order to guarantee that the extended monitor is noninterferent one must impose some constraints on the API plugins that can be invoked. Definitions 6.3 and 6.4 formalise these requirements. Definition 6.3 states that an API plugin is *confined* if it only creates/updates resources whose levels are higher than or equal to the level of the values that were used to decide which API plugin to apply. Since only the first two arguments of an API plugin are used by the API register to determine which API to apply, only the levels of these two arguments are used in the definition of confinement for APIs. In other words, an API plugin is confined if it never modifies observable resources when either one of it first two arguments is not observable.

The design of the Extensible Core JavaScript monitor guarantees that the levels of all the arguments of an API plugin are always higher than or equal to the level of the context in which that API plugin is called. Hence, we conclude that confined API plugins do neither create nor update observable resources when invoked within unobservable contexts.

Definition 6.3 (Confined Labelled API Plugin)**.** *A monitored API plugin* $(\mathsf{pg}, \mathsf{pg}_{lab}) \in \mathcal{P} \times \mathcal{P}_{lab}$ *of an API* $\mathsf{API}_{IF} = \langle \mathcal{S}, \mathcal{S}_{lab}, \mathcal{P}, \mathcal{P}_{lab}, \mathcal{R}_{IF}, \sim_{api}\rangle$ *is said to be confined, if for any API state* $\nu \in \mathcal{S}$, *API labelling* $\Xi \in \mathcal{S}_{lab}$, *sequence of values* $\overrightarrow{v}$, *sequence of levels* $\overrightarrow{\sigma}$, *security level* σ, *and annotation* α, *such that:*

- $\langle \nu, \overrightarrow{v}\rangle^\alpha \ \mathsf{pg}\ \langle \nu', v\rangle^\beta$,
- $\langle \Xi, \overrightarrow{\sigma}\rangle^\beta \ \mathsf{pg}_{lab}\ \langle \Xi', \sigma'\rangle$, *and*
- $\overrightarrow{\sigma}(0) \sqcup \overrightarrow{\sigma}(1) \not\sqsubseteq \sigma$;

It holds that: $\nu, \Xi \ \sim^\sigma_{api} \ \nu', \Xi'$ *and* $\sigma' \not\sqsubseteq \sigma$.

Definition 6.4 states that an API plugin is *noninterferent* if whenever it is executed in two low-equal API states, it produces two low-equal API states and either the two return values are both visible and coincide or they are both invisible.

Definition 6.4 (Noninterferent Labelled API plugin)**.** *A monitored API plugin* $(\mathsf{pg}, \mathsf{pg}_{lab}) \in \mathcal{P} \times \mathcal{P}_{lab}$ *of an API* $\mathsf{API}_{IF} = \langle \mathcal{S}, \mathcal{S}_{lab}, \mathcal{P}, \mathcal{P}_{lab}, \mathcal{R}_{IF}, \sim_{api}\rangle$ *is said to be noninterferent, written* $\mathbf{NI}(\mathsf{API}_{IF}, \mathsf{pg}, \mathsf{pg}_{lab})$, *if it is confined and for any two API states* ν_0 *and* ν_1 *and labellings* Ξ_0 *and* Ξ_1, *two sequences of values* $\overrightarrow{v}_0$ *and* $\overrightarrow{v}_1$ *labelled by* $\overrightarrow{\sigma}_0$ *and* $\overrightarrow{\sigma}_1$, *and security level* σ *such that:*

- $\overrightarrow{v}_0, \overrightarrow{\sigma}_0 \sim_\sigma \overrightarrow{v}_1, \overrightarrow{\sigma}_1$,
- $\nu_0, \Xi_0 \sim^\sigma_{api} \nu_1, \Xi_1$,
- $\langle \nu_0, \overrightarrow{v}_0 \rangle^\alpha$ pg $\langle \nu'_0, v_0 \rangle^\beta$ *and* $\langle \Xi_0, \overrightarrow{\sigma}_0 \rangle^\beta$ pg_{lab} $\langle \Xi'_0, \sigma_0 \rangle$, *and*
- $\langle \nu_1, \overrightarrow{v}_1 \rangle^\alpha$ pg $\langle \nu'_1, v_1 \rangle^{\beta'}$ *and* $\langle \Xi_1, \overrightarrow{\sigma}_1 \rangle^{\beta'}$ pg_{lab} $\langle \Xi'_1, \sigma_1 \rangle$;

It holds that: $\nu'_0, \Xi'_0 \sim^\sigma_{api} \nu'_1, \Xi'_1$ *and* $v_0, \sigma_0 \sim_\sigma v_1, \sigma_1$.

An API $\mathsf{API}_{IF} \in \mathcal{A}_{lab}$ is said to be confined/noninterferent if all the monitored plugins in the range of its register are confined/noninterferent.

Definition 6.5 (Confined/Noninterferent API)**.** *A monitored API* $\mathsf{API}_{IF} \in \mathcal{A}_{lab}$ *is said to be:*

- **confined** *if every API plugin* $(\mathsf{pg}, \mathsf{pg}_{lab}) \in rng(\mathsf{API}_{IF}.\mathsf{Reg})$*, verifies confinement for API plugins;*
- **noninterferent**, *written* $\mathbf{NI}(\mathsf{API}_{IF})$*, if every API plugin* $(\mathsf{pg}, \mathsf{pg}_{lab}) \in rng(\mathsf{API}_{IF}.\mathsf{Reg})$*, verifies* $\mathbf{NI}(\mathsf{API}_{IF}, \mathsf{pg}, \mathsf{pg}_{lab})$.

The next chapter presents an API for creating and manipulating dynamic tree structures that we call Core DOM and which is meant to model the Core Level 1 DOM API. The proposed API is proven to be confined and noninterferent.

6.2.3 Soundness

This section presents the two main security properties of the extensible monitor:

- Lemma 6.1 states that the extended monitor obtained by plugging a confined monitored API into the extensible monitor is also confined.
- Theorem 6.1 states that the extended monitor obtained by plugging a noninterferent monitored API into the extensible monitor is also noninterferent.

The **proofs** of the results can be found in **Appendix C**.

Lemma 6.1 (Confinement - Extensible Monitor)**.** *For any confined API* $\mathsf{API}_{IF} \in \mathcal{A}_{lab}$*, it holds that:* $\Downarrow_{IF}^{\mathsf{API}_{IF}}$ *is confined.*

Theorem 6.1 (Noninterference - Extensible Monitor)**.** *For any noninterferent and confined API* $\mathsf{API}_{IF} \in \mathcal{A}_{lab}$*, it holds that:* $\mathbf{NI}(\Downarrow_{IF}^{\mathsf{API}_{IF}})$.

Proofs are given in **Appendix C**.

Observe that an API can be noninterferent without being confined. However, in order to guarantee that the plugging of an API into the extensible monitor yields a noninterferent monitor that API must be both confined and noninterferent. This topic is further discussed in Section 6.4.

6.3 Related Work

Security of Web APIs Taly et al. [Taly 2011] proposed a static mechanism to verify API confinement in client-side JavaScript programs. More concretely, the authors designed a static analysis to formally verify whether, when integrating the code of an API into an arbitrary page, the integrator code cannot interact with the API in order to cause it to leak its internal confidential resources. Note that in [Taly 2011], the term API only refers to JavaScript libraries

whose code is explicitly included by the programmer and, therefore, is available for both runtime and static analysis. Moreover, this work aims at a very specific security property: **protecting the confidential resources of the API**. In contrast, our work aims at enforcing noninterference, which is a more expressive property than API confinement as defined in [Taly 2011]. This means that the extensible Core JavaScript monitor could be used to enforce this type of API confinement. However, that would require: **(1)** the runtime monitoring of the integrator code and **(2)** the specification of the monitored versions of the plugins exposed by the API whose internal resources are to be protected.

Recently, Hedin et al. [Hedin 2014] proposed a security enhanced JavaScript interpreter for fine-grained tracking of information flow in the presence of both JavaScript libraries and external APIs provided by the browser, such as the DOM API. In order to cater for the possible invocation of API plugins, the authors introduce the informal concept of *information-flow models* for libraries. Concretely, they consider two types of such models:

- *Shallow models* that capture the behaviour of APIs that do not have an internal state, such as the API provided by the JavaScript `Math` object (used by programs to perform mathematical tasks);
- *Deep models* that capture the behaviour of APIs that have an internal state, which, therefore, can be used to encode illegal implicit flows (such as the DOM API).

Our definition of monitored API can be seen as a formalisation of the notion of *deep information flow model* for libraries. Indeed, despite taking into account a vast number of APIs (including some functionalities of the DOM API as well as several JavaScript built-in objects), the authors of [Hedin 2014] do not present a formal framework for reasoning about extensible monitors and information flow models for libraries in a modular way.

Extensible Semantics In recent years, interoperability has emerged as a central feature in programming language design and implementation. This fact has pushed forward research on runtime mechanisms for allowing programs written in different languages to interact with each other in a seamless way [Ramsey 2011]. However, the topic of formal methods specifically designed to reason about *language extension* remains relatively unexplored. Matthews and Findler [Matthews 2009] presented a formal semantics for reasoning about multi-language programs. In order to cater for the interaction between sub-programs written in different languages (but viewed as parts of the same global program), the authors proposed a technique based on simple "cross-language casts that regulate both control flow and value conversion between languages". In other words, the combined languages must be extended with special *boundary* operators that serve to: **(1)** identify which regions of the global program are written in which language and **(2)** perform the required conversions between the values of both languages. This technique can be used to model a restricted form of interaction between Core JavaScript programs and external APIs. More precisely, one can model an external API as a language and then plug the API-language into the Core JavaScript runtime using a *boundary operator*. However, this would yield a more restrictive (and less realistic) mechanism for interaction between Core JavaScript programs and external APIs.

In the context of Aspect oriented programming [Kiczales 1997], Djoko et al. [Djoko 2008] proposed a formal semantics for performing the runtime *weaving* of arbitrary aspects into a simple imperative language. Intuitively, an *aspect* can be seen as function that given a program configuration produces a new configuration. Hence, aspects can modify both the memory and the syntax of the program that is executing (many times distorting its semantics in unpredictable ways). The authors of [Djoko 2008] define three categories of aspects and identify, for each category, a class of temporal properties preserved by the weaving of the aspects of that category.

$$\textsc{Property Look-up}$$
$$\frac{\langle \hat{e}_0 \mid j \rangle = \mathcal{C}\langle e_0 \rangle \qquad \langle \hat{e}_1 \mid k \rangle = \mathcal{C}\langle e_1 \rangle \qquad \langle \hat{e}_0, \hat{e}_1, \hat{e} \mid i \rangle = \mathcal{C}\langle e_0[e_1]^i \rangle \qquad \hat{e} = \begin{cases} \hat{e}_0, \\ \hat{e}_1, \\ \texttt{\#tmp} = \texttt{\$register}(\$v_j, \$v_k)\ ? \\ \quad \big(\ \texttt{\#tmp.check}(\$l_j, \$l_k, \$v_j, \$v_k), \\ \quad \$v_i = \$v_j[\$v_k], \\ \quad \$l_i = \texttt{\#tmp.label}(\$v_i, \$l_j, \$l_k, \$v_j, \$v_k)\big) \\ \quad :\ (\hat{e}) \end{cases}}{\mathcal{C}_{api}\langle e_0[e_1]^i \rangle = \langle \hat{e}_0, \hat{e}_1, e' \mid i \rangle}$$

$$\textsc{Method Call}$$
$$\frac{\langle \hat{e}_0 \mid j \rangle = \mathcal{C}\langle e_0 \rangle \qquad \langle \hat{e}_1 \mid k \rangle = \mathcal{C}\langle e_1 \rangle \qquad \langle \hat{e}_2 \mid l \rangle = \mathcal{C}\langle e_2 \rangle \qquad \langle \hat{e}_0, \hat{e}_1, \hat{e}_2, \hat{e} \mid i \rangle = \mathcal{C}\langle e_0[e_1](e_2)^i \rangle \qquad \hat{e} = \begin{cases} \hat{e}_0, \\ \hat{e}_1, \\ \hat{e}_2, \\ \texttt{\#tmp} = \texttt{\$register}(\$v_j, \$v_k)\ ? \\ \quad \big(\ \texttt{\#tmp.check}((\$l_j, \$l_k, \$l_l, \$v_j, \$v_k, \$v_l), \\ \quad \$v_i = \$v_j[\$v_k](\$v_l), \\ \quad \$l_i = \texttt{\#tmp.label}(\$v_i, \$l_j, \$l_k, \$l_l, \$v_j, \$v_k, \$v_l)\big) \\ \quad :\ (\hat{e}) \end{cases}}{\mathcal{C}_{api}\langle e_0[e_1](e_2)^i \rangle = \langle \hat{e}_0, \hat{e}_1, \hat{e}_2, e' \mid i \rangle}$$

Figure 6.3: Extended Compiler - $\mathcal{C}_{API}$

Interestingly, API plugins can be viewed as *observer aspects*. Observer aspects are aspects that "do not modify the base program's state and control flow." In fact, the mechanism proposed in this thesis for extending the semantics of Core JavaScript is very similar to the one proposed in [Djoko 2008]. The main difference between the two mechanisms is that, in our case, the API register selects which plugin to execute based on the values of the first two subexpressions of the intercepted expression, whereas in [Djoko 2008] the domain of the *aspect weaver* is the entire program configuration. Moreover, while aspects can change intercepted configurations in arbitrary ways, API plugins can neither change the Core JavaScript memory nor the syntax of the program that is executing. However, the additional expressivity of the *aspect weaving mechanism* is not needed for the concrete case of Web APIs, which interact with the JavaScript runtime in a more restricted way.

6.4 Discussion

6.4.1 Toward the Inlining of Extensible Information Flow Monitors

This section presents an informal description of a methodology for the inlining of monitors extended with arbitrary APIs. A call to an API plugin cannot be instrumented in the same way one instruments a normal JavaScript method call, simply because the code of API plugins is usually not available for instrumentation. Assuming that API labellings are instrumented in memory, we propose to associate each API plugin with three special JavaScript methods – *domain*, *check* and *label*, called the *IFlow Signature* of the API plugin. Each one of these methods, serves a different purpose:

- *domain* checks whether or not to apply the API plugin.

- *check* checks whether the constraints associated with the API plugin are verified,
- *label* updates the instrumented labelling and outputs the reading effect associated with a call to the API plugin.

The functions *check* and *label* must be specified separately because *check* has to be executed before calling the plugin (in order to prevent its execution when it can potentially trigger a security violation), whereas *label* must be executed after calling the plugin (so that it can label the memory resulting from its execution).

This approach to the inlining of extensible security monitors also requires the existence of a runtime function that simulates the API Register, which is assumed to be bound to the global variable `$register`. The function bound to `$register` makes use of the methods *domain* of each plugin to decide whether the current method call or property look-up triggers the invocation of an API plugin, in which case it returns an object containing the corresponding IFlow Signature (otherwise it simply returns null).

Figure 6.3 presents the extension of the inlining compiler introduced in Chapter 4 that takes into account the possible invocation of external APIs. We denote the new compiler by $\mathcal{C}_{API}$. This compiler coincides with the previous one for every program construct with the exception of method calls and property look-ups, in which case it has to take into account the possible invocation of external APIs. For these two constructs, the code generated by the compiler proceeds as follows:

1. It executes the statements corresponding to the compilation of its subexpressions;
2. It checks, using the values of to the first two subexpressions, whether that property look-up or method call is associated with an IFlow signature (using the function bound to `$register`);
3. And, finally, it does one of the following:
   - If the call to the function bound to `$register` returns an IFlow signature, the compiled program:
     - executes the method *check* of the IFlow signature,
     - executes an expression obtained from the original one by replacing its subexpressions with the bookkeeping variables that hold their current values,
     - executes the method *label* of the IFlow signature in order to update the generated memory and to obtain the reading effect of the call to that API.
   - If the call to the function bound to `$register` returns null, the compiled program acts as the compilation of the original program using the nonextensible inlining compiler.

6.4.2 Further Comments on Confinement for APIs

While it is true that the definition of noninterference for API plugins is the standard inductive invariant needed to prove noninterference in dynamic monitors [Austin 2009], the definition of confinement is not that usual. In fact, the definition of confinement for APIs depends on the features of the proposed mechanism for plugging APIs into the language runtime. Different mechanisms require different confinement properties.

The extensible semantics selects which API plugin to execute depending on the values of the first two subexpressions of the intercepted expression. Hence, in order to follow the no-sensitive-upgrade discipline, the selected plugin can only change its internal resources whose levels are higher than or equal to the levels of its first two arguments. Suppose, however, that one changes the proposed mechanism so that the values of all the subexpressions of the intercepted expression

are used to determine which API plugin to execute. In this case, the confinement property should state that the selected plugin can only change its internal resources whose levels are higher than or equal to the *lub* between the levels of all of its arguments. In a nutshell: the more flexible is the mechanism for plugging APIs into the language runtime, the more restrictive must be the confinement property that the plugged APIs verify.

Chapter 7

Monitoring Secure Information Flow in a DOM-like API

Contents

Interaction between client-side JavaScript programs and the HTML document is done *via* the DOM API [W3C Recommendation 2005]. In contrast to the ECMA Standard [5th edition of ECMA 262 2011] that specifies in full detail the internals of objects created during the execution of JavaScript programs, the DOM API only specifies the behaviour that DOM interfaces are supposed to exhibit when a program interacts with them. Hence, browser vendors are free to implement the DOM API as they see fit. In fact, in all major browsers, the DOM is not managed by the JavaScript engine but by a separate engine, often called the *render engine* [Grosskurth 2005], especially dedicated to that purpose. Therefore, the design of an information flow monitor for client-side JavaScript Web applications must take into account the DOM API.

This chapter presents a formal monitored API (check Chapter 6) called Core DOM. The Core DOM API is an API for creating and manipulating dynamic tree structures, while at the same time enforcing secure information flow. This formal API models an important fragment of the DOM Core Level 1 API, that includes references and live collections. Furthermore, Core DOM is proven to be noninterferent according to Definition 6.4. Hence, as we have seen in Chapter 6, the plugging of Core DOM into the extensible Core JavaScript monitor yields a noninterferent monitor.

Russo et al. [Russo 2009] were the first to study the problem of information flow control in dynamic tree structures, for a model where programs are assumed to operate on a single current working node at a time. However, in real client-side JavaScript, tree nodes are first-class values, which means that a program can store in memory several references to different nodes in the DOM forest at the same time. In contrast to the model of [Russo 2009], in Core DOM, tree nodes are treated as first-class values and thus they support operations available to other types of values, such as assignment to variables. Interestingly, this language design feature enables us to implement a more fine-grained information flow control mechanism than previous approaches (discussed in detail in Section 7.4), since it becomes possible to distinguish the security level of the node itself from both the security level of the value that is stored in the node and from the level of its position in the DOM forest.

Live collections are a special kind of data structures featured in the DOM Core Level 1 API that automatically and dynamically reflect the changes that occur in the document. There are several types of live collections. For instance, the method `getElementsByTagName` returns the live collection that contains the DOM nodes with the *tag name* given as input. In the following example, after retrieving the initial collection of **DIV** nodes, the program iterates over the *current* size of this collection, while introducing a new **DIV** node at each step:

```
divs = document.getElementsByTagName("DIV"),
i = 0,
while (i <= divs.length) {
   document.appendChild(document.createElement("DIV")),
   i = i + 1
}
```
(7.1)

Every time a new **DIV** node is inserted in the *document* (no matter where in its structure), it is also inserted in the live collection bound to `divs`. Due to the live update of the loop condition, if the initial *document* contains at least one **DIV** node, the program does not terminate.

Live collections can be exploited to encode new types of information leaks. Therefore, we include in the Core DOM API several plugins for creating and traversing live collections in tree structures. We further demonstrate that these plugins effectively augment the observational power of an attacker and we show how to monitor their execution in order to preserve noninterference.

Outline This chapter is structured as follows: Section 7.1 formally introduces the targeted Core DOM API. Section 7.2 begins with a discussion of the challenges of controlling information flow in Core DOM and, then, presents its monitored version. Section 7.3 extends the monitored version of the Core DOM API with additional plugins for the secure creation and manipulation of live collections. Section 7.4 presents a discussion of the related work. Finally, in Section 7.5, we analyse the following questions: **(1)** How are the newly identified types of leaks present in the real DOM API? **(2)** How is the proposed monitoring mechanism more precise than that of [Russo 2009]?

7.1 Core DOM

The DOM data structure can be viewed as a forest of DOM nodes containing a special tree that corresponds to the *document* being displayed by the browser. Indeed, most of the information flows that are specific to the DOM API have to do with dynamic tree operations [Russo 2009], such as the creation, insertion, or removal of DOM nodes. Hence, in Core DOM, we include the most relevant methods and properties of the DOM Core Level 1 API used for traversing and

$r_0, \cdots, r_i \in \mathtt{R}_{DOM} \subset \mathtt{Ref}$			% Core DOM References
$\#doc \in \mathtt{R}_{DOM}$			% Document Reference
$n \in \mathtt{Node} \ni \mathtt{doc}$	$::=$	$\langle m, v, r, \overrightarrow{r} \rangle$	% Core DOM Node
$f \in \mathtt{F}_{DOM}$	$::=$	$[\#doc \mapsto \mathtt{doc}, r_0 \mapsto n_0, \cdots, r_i \mapsto n_i]$	% Core DOM Forest
$\mathsf{dplug} \in \mathcal{P}^{DOM}$			% Core DOM Plugins
$\mathcal{R}^{DOM} : \mathtt{Ref} \times \mathtt{Prim} \to \mathcal{P}^{DOM}$			% Core DOM Register
$\mathsf{CoreDOM} \in \mathcal{A}$		$\langle \mathtt{F}_{DOM}, \mathcal{P}^{DOM}, \mathcal{R}^{DOM} \rangle$	% Core DOM API

Table 7.1: Components of the Core DOM API Formal Model

updating tree structures. Concretely, in Core DOM, every node has a type, called its *tag* (for instance, **DIV**) and can store a single value taken from `Prim`. All the nodes in memory form a *forest*. Hence, every node has a possibly empty list of *children* and at most a single *parent*. A node with no parent is called a *root* node, while a node with no children is called a *leaf node*. Nodes with the same parent are called *siblings*. Whenever a node has a parent, the position that it occupies in the list of children of its parent is called the *index* of the node. For instance, if a node n_1 is the first child of a given node n_0, the index of n_1 is 0.

The Core DOM API is assumed to be available through a special reference $\#doc$ bound to the global variable `document` and to expose the following methods and properties:

- `document.createElement(tag)`: creates a new node with tag name `tag`;
- `node0.appendChild(node1)`: appends `node1` to the list of children of `node0`, provided that `node1` is a root node;
- `node0.removeChild(node1)`: removes `node1` from the list of children of `node0`, provided that `node1` is indeed a child of `node0`;
- `node`$[i]$: evaluates to the i+1$^{\text{th}}$ child of `node`, provided that it has at least `i` $+ 1$ children;
- `node.length`: evaluates to the number of children of `node`;
- `node.parentNode`: retrieves the parent of `node`;
- `node.nodeValue`: retrieves the value that is stored in `node`;
- `node.store(value)`: stores `value` inside `node`.

One important aspect in which the Core DOM API differs from the real DOM specification [W3C Recommendation 2005] is that, in Core DOM, the child nodes of a given DOM node are directly accessed through that node. Instead, in the real DOM specification, the child nodes of a given node are accessed through a special object bound to the node's property `"childNodes"`. This object behaves as a list that contains the child nodes of the given node. Hence, instead of writing `div1.childNodes`$[i]$ to access the $i + 1^{\text{th}}$ child of the **DIV** node bound to `div1`, we simply write `div1`$[i]$.

7.1.1 Core DOM - Formal Model

As discussed in Chapter 6, a formal API is modelled as a triple of the form $\langle \mathcal{S}, \mathcal{P}, \mathcal{R} \rangle$ consisting of a set of API states, a set of plugins that operate on those states, and an API register. Hence, the Core DOM API is formally modelled as triple the $\langle \mathtt{F}_{DOM}, \mathcal{P}^{DOM}, \mathcal{R}^{DOM} \rangle$, where $\mathtt{F}_{DOM}$

$$\mathcal{R}_{IF}^{DOM}(r_0, v_1) = \begin{cases} (\mathsf{new}, \mathsf{new}_{lab}) & \text{if } r_0 = \#doc \ \wedge \ v_1 = \texttt{"createElement"} \\ (\mathsf{append}, \mathsf{append}_{lab}) & \text{if } r_0 \in \mathtt{R}_{DOM} \ \wedge \ v_1 = \texttt{"appendChild"} \\ (\mathsf{remove}, \mathsf{remove}_{lab}) & \text{if } r_0 \in \mathtt{R}_{DOM} \ \wedge \ v_1 = \texttt{"removeChild"} \\ (\mathsf{item}, \mathsf{item}_{lab}) & \text{if } r_0 \in \mathtt{R}_{DOM} \ \wedge \ v_1 \in \mathtt{Num} \\ (\mathsf{length}, \mathsf{length}_{lab}) & \text{if } r_0 \in \mathtt{R}_{DOM} \ \wedge \ v_1 = \texttt{"length"} \\ (\mathsf{parent}, \mathsf{parent}_{lab}) & \text{if } r_0 \in \mathtt{R}_{DOM} \ \wedge \ v_1 = \texttt{"parentNode"} \\ (\mathsf{value}, \mathsf{value}_{lab}) & \text{if } r_0 \in \mathtt{R}_{DOM} \ \wedge \ v_1 = \texttt{"nodeValue"} \\ (\mathsf{store}, \mathsf{store}_{lab}) & \text{if } r_0 \in \mathtt{R}_{DOM} \ \wedge \ v_1 = \texttt{"storeValue"} \end{cases}$$

Figure 7.1: The Core DOM Monitored API Register

is the set of Core DOM states, called *forests*, $\mathcal{P}^{DOM}$ is the set of Core DOM plugins, and $\mathcal{R}^{DOM} : \mathtt{Ref} \times \mathtt{Prim} \to \mathcal{P}^{DOM}$ is the Core DOM register. In the following, we use CoreDOM to refer to the Core DOM API.

A Core DOM forest $f \in \mathtt{F}_{DOM} : \mathtt{R}_{DOM} \to \mathtt{Node}$ is as a partial mapping from a set $\mathtt{R}_{DOM}$ of DOM references to the set Node of DOM nodes. A DOM node is a tuple of the form: $\langle m, v, r, \overrightarrow{r} \rangle$, where: **(1)** m is the node's tag, **(2)** v the value it stores, **(3)** r the reference pointing to its parent, and **(4)** $\overrightarrow{r}$ its list of children (more precisely, a list of references, each pointing to one of its children). For simplicity, given a DOM node $n \in \mathtt{Node}$, we denote by $n.\mathsf{tag}$, $n.\mathsf{value}$, $n.\mathsf{parent}$, and $n.\mathsf{children}$ its tag, value, parent, and list of children, respectively.

As discussed in Chapter 6, the formal semantics of the Core DOM API assumes that the set of references used by this API does not overlap with the set of references used by the semantics of Core JavaScript. While Core DOM references are used for the allocation of Core DOM nodes, Core JavaScript references are used for the allocation of Core JavaScript objects. Hence, the Core DOM node allocator, $\mathsf{fresh}_{DOM} : \mathcal{L} \to \mathtt{R}_{DOM}$ is assumed to generate references in a set that does not overlap with the set used for the same purpose by the allocator of Core JavaScript. Furthermore, we assume that every Core DOM forest contains a special Core DOM node $\mathtt{doc} \in \mathtt{Node}$ that is the root of the tree corresponding to the *document* displayed by the browser. The *document* node doc is pointed to by a fixed reference $\#doc \in \mathtt{R}_{DOM}$. The components of the Core DOM API formal model are summarised in Table 7.1.

The API register is used by the extended semantics to determine in which conditions a method call or a property look-up should be handled by the API that is plugged into the extensible monitor. Since the set of references used by the Core DOM API does not overlap with the set of references used by the Core JavaScript semantics, the DOM register can easily identify the conditions under which a property look-up or a method call should trigger the execution of a Core DOM plugin. Intuitively, that should happen when a program inspects a property of a Core DOM node or when a program performs a method call on a Core DOM node. In both cases, the string corresponding to the inspected property or to the name of the method determines which Core DOM plugin is to be executed. In order to avoid repetition, we omit the specification of the Core DOM API register. Instead, Figure 7.1 presents its monitored version $\mathcal{R}_{IF}^{DOM} : \mathtt{Ref} \times \mathtt{Prim} \to \mathcal{P}^{DOM} \times \mathcal{P}_{lab}^{DOM}$. To the obtain the unmonitored register from its monitored version, it suffices to ignore the second component of the output. The conditions under which each Core DOM plugin is executed are explained below.

- When a program invokes a method named "createElement" on the *document* node doc (which is pointed to by $\#doc$), the Core DOM plugin new is executed.
- When a program invokes a method named "appendChild" on a Core DOM node, the Core DOM plugin append is executed.

- When a program invokes a method named `"removeChild"` on a Core DOM node, the Core DOM plugin remove is executed.
- When a program inspects an integer property of a Core DOM node, the Core DOM plugin item is executed.
- When a program inspects the property `"length"` of a Core DOM node, the Core DOM plugin length is executed.
- When a program inspects the property `"parentNode"` of a Core DOM node, the Core DOM plugin parent is executed.
- When a program inspects the property `"nodeValue"` of a Core DOM node, the Core DOM plugin value is executed.
- When a program invokes a method named `"storeValue"` on a Core DOM node, the Core DOM plugin store is executed.

It is worth emphasising that the *document node* `doc` plays the role of the *entry point* to the Core DOM API. Indeed, programs use the *document node* to create Core DOM nodes. Then, they can interact with these nodes directly using the plugins that they expose.

The specification of the Core DOM plugins makes use of a semantic function $\mathsf{Ancestor} : \mathtt{R}_{DOM} \times \mathtt{F}_{DOM} \rightarrow 2^{\mathtt{Node}}$, formally given in Definition 7.1, that, given a node reference r and a forest f, outputs a the set that contains all the ancestors of the node pointed to by r in f (which contains the node itself). Hence, $r_1 \in \mathsf{Ancestor}(r_0, f)$ means that the node pointed to by r_1 is an ancestor of the node pointed to by r_0 in f.

Definition 7.1 (Ancestor)**.** *The function* $\mathsf{Ancestor} : \mathtt{R}_{DOM} \times \mathtt{F}_{DOM} \rightarrow 2^{\mathtt{Node}}$ *is defined as follows:*

$$\mathsf{Ancestor}(r, f) = \begin{cases} \{r\} & \text{if } f(r).\mathsf{parent} = \mathsf{null} \\ \{r\} \cup \mathsf{Ancestor}(f(r).\mathsf{parent}, f) & \text{otherwise} \end{cases}$$

The formal specification of the plugins that compose the Core DOM API is presented in Figure 7.2.[1] As described in Chapter 6, each DOM API plugin is modelled as a relation of the form $\langle f, \overrightarrow{v} \rangle^{\alpha}$ dplug $\langle f', v \rangle^{\beta}$, where: **(1)** f is the Core DOM forest on which the API plugin is invoked, **(2)** $\overrightarrow{v}$ the sequence of values that it receives as input, **(3)** α an arbitrary annotation to be used by the programmer to provide additional information to the monitor plugin, **(4)** f' the forest that results from the execution of the plugin, **(5)** v the return value of the plugin, and **(6)** β an internal event generated by the plugin for the use of its monitor counterpart. The Core DOM API plugins are briefly described below.

- The plugin new expects as arguments: **(1)** the reference *#doc* pointing to the *document* node `doc`, **(2)** the string `"createElement"`, and **(3)** the tag name m of the node to be created. The plugin first creates a new node, which is allocated in a new node reference r, computed using the allocator fresh_{DOM}. The explanation of the argument given to the allocator is presented in Section 7.2, where the mechanism for labelling Core DOM nodes is introduced. The tag component of the newly created node is set to m. Both the value and the parent components of the new node are set to null. Finally, the children component is set to the empty sequence, as newly created nodes do not have any child nodes. The plugin return value is the reference of the new node.

[1] The specification makes use of the operators introduced in Chapter 2 for the manipulation of sequences of values.

$$\textsc{New}\quad \frac{r = \mathsf{fresh}_{DOM}(\sigma_0) \qquad f' = f\,[r \mapsto \langle m, \mathsf{null}, \mathsf{null}, \varepsilon\rangle]}{\langle f, \#doc :: \texttt{"createElement"} :: m\rangle^{(\sigma_0,\sigma_1,\sigma_2)}\ \mathsf{new}\ \langle f', r\rangle^{(r,\sigma_0,\sigma_1,\sigma_2)}}$$

$$\textsc{Append}\quad \frac{\begin{array}{c} r' \notin \mathsf{Ancestor}(r, f) \qquad f(r) = \langle m, v, \hat{r}, \overrightarrow{r}\rangle \qquad f(r') = \langle m', v', \mathsf{null}, \overrightarrow{r'}\rangle \\ \overrightarrow{r} = \varepsilon \Rightarrow r'' = \mathsf{null} \qquad \overrightarrow{r} \neq \varepsilon \Rightarrow r'' = \overrightarrow{r}.\mathsf{last} \\ f' = f\left[r \mapsto \langle m, v, \hat{r}, \overrightarrow{r} :: r'\rangle, r' \mapsto \langle m', v', r, \overrightarrow{r'}\rangle\right] \end{array}}{\langle f, r :: \texttt{"appendChild"} :: r'\rangle\ \mathsf{append}\ \langle f', r'\rangle^{(r,r',r'')}}$$

$$\textsc{Remove}\quad \frac{\begin{array}{c} f(r) = \langle m, v, \hat{r}, \overrightarrow{r}\rangle \qquad f(r).\mathsf{children}(i) = r' \qquad f(r') = \langle m', v', r, \overrightarrow{r'}\rangle \\ f' = f\left[r \mapsto \langle m, v, \hat{r}, \mathsf{Shift}_{\mathsf{L}}(\overrightarrow{r}, i)\rangle, r' \mapsto \langle m', v', \mathsf{null}, \overrightarrow{r'}\rangle\right] \end{array}}{\langle f, r :: \texttt{"removeChild"} :: r'\rangle\ \mathsf{remove}\ \langle f', r'\rangle^{(r,r')}}$$

$$\textsc{Item}\quad \frac{f(r).\mathsf{children}(i) = r'}{\langle f, r :: i\rangle\ \mathsf{item}\ \langle f, r'\rangle^{(r')}} \qquad \textsc{Length}\quad \frac{i = |f(r).\mathsf{children}|}{\langle f, r :: \texttt{"length"}\rangle\ \mathsf{length}\ \langle f, i\rangle^{(r)}}$$

$$\textsc{Parent}\quad \frac{f(r).\mathsf{parent} = v}{\langle f, r :: \texttt{"parentNode"}\rangle\ \mathsf{parent}\ \langle f, v\rangle^{(r)}} \qquad \textsc{Value}\quad \frac{f(r).\mathsf{value} = v}{\langle f, r :: \texttt{"nodeValue"}\rangle\ \mathsf{value}\ \langle f, v\rangle^{(r)}}$$

$$\textsc{Store}\quad \frac{\begin{array}{c} f(r) = \langle m, v, \hat{r}, \overrightarrow{r}\rangle \\ f' = f\left[r \mapsto \langle m, v', \hat{r}, \overrightarrow{r}\rangle\right] \end{array}}{\langle f, r :: \texttt{"storeValue"} :: v'\rangle\ \mathsf{store}\ \langle f', v'\rangle^{(r)}}$$

$$\textsc{Core DOM Plugins}\quad \mathcal{P}^{DOM} = \left\{\ \mathsf{new}, \mathsf{append}, \mathsf{remove}, \mathsf{item}, \mathsf{length}, \mathsf{parent}, \mathsf{value}, \mathsf{store}\ \right\}$$

Figure 7.2: Core DOM API Plugins

- The plugin append expects as arguments: **(1)** the reference r of the node to which another node is to be appended, **(2)** the string `"appendChild"`, and **(3)** the reference r' of the node to be appended. It then appends the node pointed to by r' to the list of children of the node pointed to by r, provided that the node pointed to by r' is not an ancestor of the node pointed to by r in the Core DOM forest. This constraint prevents the forming of cycles in the DOM forest. The plugin generates an internal event consisting of a 3-tuple that contains r, r', and the reference of the new left sibling of the node pointed to by r'. Finally, the plugin return value is r'.

- The plugin remove expects as arguments: **(1)** the reference r of the node from which another node is to be removed, **(2)** the string `"removeChild"`, and **(3)** the reference r' of the node to be removed. Provided that the node pointed to by r' is a child of the node

pointed to by r, the plugin removes r' from the list of children of the node pointed to by r. Additionally, it sets the parent component of the node pointed to by r' to null, since, after removing this node from the list of children of its parent, it becomes a root node. Finally, the plugin return value is r'.

- The plugin item expects as arguments: the reference r of the node whose child is to be inspected and the position i that that child occupies in the list of children of the node pointed to by r. The plugin return value is the $i + 1^{\text{th}}$ reference in the list of children of the node pointed to by r (provided that such reference exists).

- The plugin length expects as arguments: the reference r of the node whose number of children is to be inspected and the string `"length"`. The plugin return value is the number of children of the node pointed to by r.

- The plugin parent expects as arguments: the reference r of the node whose parent is to be inspected and the string `"parentNode"`. The plugin return value is the parent component of the node pointed to by r. Hence, if the node pointed to by r is not a root node, the return of the plugin is the reference that points to its parent. Otherwise, the plugin returns null.

- The plugin value expects as arguments: the reference r of the node whose value is to be inspected and the string `"nodeValue"`. The plugin returns the value stored in the node pointed to by r (which corresponds to its component value).

- The plugin store expects as arguments: **(1)** the reference r of the node whose stored value is to be updated, **(2)** the string `"storeValue"`, and **(3)** the new value v to be stored in the node pointed to by r. The plugin simply sets the value component of the node pointed to by r to v and returns v.

7.2 Monitoring Secure Information Flow in the Core DOM API

Before proceeding to the description of the monitor plugins for securing information flow in the Core DOM API, we discuss the main challenges imposed by the particular features of this API and how we propose to tackle them.

7.2.1 Challenges for Information Flow Control in Core DOM

The range of tree operations offered by the Core DOM API allows information to be stored in and inspected from Core DOM nodes in several ways:

- A node can be created and its existence tested;

- A value can be stored in a node and later read from that node;

- A node can be inserted at/removed from a certain *position* of the Core DOM forest and its new *position* can be later checked;

- A node can be inserted at/removed from a certain position of the Core DOM forest and the number of children of both its former parent and new parent can be retrieved afterwards.

Here, we define the *position* of a node as the pair consisting of its parent in the Core DOM forest and its index. Observe that, according to this definition, if we know the positions of all the nodes in the Core DOM forest, we can reconstruct its shape.

The operations discussed above can be used to encode security leaks via the different information components that are associated with every node. We now examine these leaks in detail and introduce the formal techniques we use for tackling them. We assume in the examples that the original forest contains three initial **DIV** nodes, bound to `div0`, `div1`, and `div2` respectively and created as follows:

$$\begin{array}{l}\texttt{div0} = \texttt{document.createElement("DIV")}^{L,L,H},\\ \texttt{div1} = \texttt{document.createElement("DIV")}^{L,H,L},\\ \texttt{div2} = \texttt{document.createElement("DIV")}^{L,H,L}\end{array} \tag{7.2}$$

The security levels that annotate the three methods calls are explained in Subsection 7.2.3.

7.2.1.1 Differentiating Information Components

Each node in a Core DOM forest can be seen to carry **four main information components**: its existence, its value, its position and its number of children. These components can be manipulated separately, and, therefore, there is value in treating them individually. In other words, it is useful to assign a different security label to each of these components. For instance, in the following program, the final position of the node bound to `div2` carries *high* information (because it is inserted in a *high* context), in spite of containing the *low* level value originally bound to `l0`.

$$\begin{array}{l}\texttt{div2.storeValue(l0)},\\ \texttt{h ? (div0.appendChild(div2)) : (div1.appendChild(div2))}\end{array} \tag{7.3}$$

After the execution of this program, the position of the node bound to `div2` should not be revealed to a low observer. Its value, however, can be made public. Hence, while the evaluation of `div2.parentNode` should yield a *high* value, the evaluation of the `div2.nodeValue` can yield a *low* value. Similarly, there is no reason why the position of a node that stores a secret value should not be public.

By treating tree nodes as first-class values, we can naturally differentiate the security levels that are associated to each of the node's information components. We propose to associate every tree node with four security levels:

- The *value level* of a node is the level of the value that it stores.
- The *position level* of a node is the level of its position in the Core DOM forest.
- The *structure security level* of a node is the level associated to that node's number of children.
- The *node level* is the level associated to the existence of the node itself. This level can be seen as the level of the context in which the node is created.

7.2.1.2 Security Leaks in the Core DOM API

When removing a node from the list of children of another node, the indexes of its right siblings change, thereby entailing a new kind of implicit flow. Consider the following example:

$$\begin{array}{l}\texttt{div0.appendChild(div1)},\\ \texttt{div0.appendChild(div2)},\\ \texttt{h ? (div0.removeChild(div1))},\\ \texttt{l0} = \texttt{div0}[0]\end{array} \tag{7.4}$$

that serves as the running example in this subsection. This program appends the nodes bound to `div1` and to `div2` to the list of children of the node bound to `div0` (which is originally empty). Then, depending on the value of the *high* variable `h`, it removes the node bound to `div1` from the list of children of the node bound to `div0`. Hence, depending on the value of `h`, the program assigns either the reference of the node bound to `div1` or the reference of the node bound to `div2` to the *low* variable `l0`. We refer to these forms of security leaks as *order leaks*, as they leverage information about the order of Core DOM nodes in the list of children of their parents.

In a nutshell, when removing one node from the list of children of another, the positions of its right siblings also change. Complementarily, when appending a node to the list of children of another node, the position it will occupy depends on the positions of its left siblings. Therefore, the monitor API enforces that, for every node in the Core DOM forest, **the position levels of its child nodes are monotonically increasing**. Hence, the position level of a node is always **(1)** higher than or equal to the position levels of its left siblings and **(2)** lower than or equal to the position levels of its right siblings.

Suppose, for instance, that: **(1)** a node n_B is removed from the list of children of a node n_A within an invisible context and that **(2)** n_B has a right sibling n_C. For the monitor to allow this removal to go through, n_B must have an invisible position level (in order to prevent the implicit flow resulting from changing the position of a node with a visible position inside an invisible context). When removing n_B, the position of n_C is also changed. However, the monitor does not have to check whether the position level of n_C is higher than or equal to the level of the context. The reason for this is that the monitor enforces the position level of n_C to be higher than or equal to the position level of n_B. Since the position level of n_B is higher than or equal to the level of the context, it follows that the position level of n_C is also higher than or equal to the level of the context. Hence, even assuming that the level of the context is invisible, the removal of n_B does not produce any visible changes. Because, in this scenario, the monitor blocks the execution if either the position level of n_B or the position level of n_C are visible.

The fact that a program can inspect the number of children of a given node can also be exploited to encode implicit information flows. If we add the assignment `l1 = div0.length` to the end of the Program 7.4, `l1` will be either set to 2 or to 1 depending on the value of the *high* variable `h`. The *structure security level* of a Core DOM node is meant to control this kind of leaks. One can look at the *structure security level* of a Core DOM node as an upper bound on the levels of the contexts in which one can add or remove nodes to or from its list of children. Hence, if a node has *low* structure security level, one cannot insert/remove nodes in/from its list of children in *high* contexts. Therefore, the level associated with looking-up the number of children of a given node corresponds to its structure security level.

7.2.1.3 Flow-sensitive versus Flow-insensitive Monitoring in Core DOM

Both the **structure security level** and **the position level** of a node are used to control the implicit flows that can be encoded by inserting/removing nodes in/from the Core DOM forest. Hence, in order to apply the no-sensitive-upgrade discipline [Zdancewic 2002], these levels **cannot be upgraded**. This point is exemplified in Table 7.2, which illustrates two pairs of monitored executions of the program shown on its left column. Each pair consists of a monitored execution that follows the *no-sensitive-upgrade* discipline and a monitored execution (called *naive*) that simply raises the levels of the resources updated in secret contexts to the levels of those contexts. Each pair consists of two executions that start from the same memory and the same forest. The initial labelled memories corresponding to each pair only differ in the value of the *high* variable `h`. In one case the *high* variable `h` is initially set to 0, whereas in the other it is set to 1. All initial memories and labellings are such that `div0` and `div1` each bind a root node with *low* structure security level. The node bound to `div0` is pointed to by r_0 and

Program:	h = 0	h = 1	
	Both Approaches	*Naive Approach*	*No-Sensitive-Upgrade*
l = true	$\Sigma_v(r \cdot \texttt{"l"}) := L$	$\Sigma_v(r \cdot \texttt{"l"}) := L$	$\Sigma_v(r \cdot \texttt{"l"}) := L$
h ?	branch not taken	branch taken	branch taken
div0.appendChild(div1)	—	$\Xi(r_0).\mathsf{struct} := H$	*stuck*
(div0.length == 0) ?	branch taken	branch not taken	—
l = false	$\Sigma_v(r \cdot \texttt{"l"}) := L$	—	—
Final Low Memory:	l = false	l = true	—

Table 7.2: The Structure Security Level of Core DOM nodes Must Be Flow Insensitive

the node bound to `div1` is pointed to by r_1 (in the four forests). The reference r points to the current active scope object. Table 7.2 shows how the Core JavaScript property-value labelling Σ_v as well as the forest labelling Ξ evolve during the two pairs of monitored executions. Since the monitored executions starting from the memory that initially maps `h` to 0 coincide, they are represented together.

Consider the monitored executions starting from the memory that initially maps `h` to 1. While the monitor following the *naive approach* raises the structure security level of the node bound to `div0` to H (allowing the execution to go through), the monitor following the *no-sensitive-upgrade* discipline blocks the execution when the program tries to append the node bound to `div1` to the list of children of the node bound to `div0` in a *high* context. Observe that, despite of executing the same program in two low-equal memories, the monitor that follows the *naive approach* generates two memories that are not low-equal.

By replacing the test of the second conditional expression with `div1.parentNode`, one obtains an analogous example that illustrates why position levels cannot be flow-sensitive. In contrast to the position level and to the structure security level, **the value level of a node can be upgraded**, since the value stored in a node is explicitly set. However, such upgrades cannot be caused by implicit information flows.

7.2.2 An Attacker Model for the Core DOM API

In order to formally characterise what part of a Core DOM forest an attacker can observe at a given security level, one must define a low-equality relation $\sim_{DOM}$ for Core DOM forests parameterizable with a security level σ. Intuitively, the low-equality relation $\sim_{DOM}$, for labelled Core DOM forests, must be such that: whenever two labelled forests are related by $\sim_{DOM}$ at a given level σ, an attacker at level σ cannot distinguish the two of them.

In order to define the low-equality for Core DOM forests, we start by defining the notion of node labelling. A node labelling $\Xi \in \mathtt{Lab}_{DOM} : \mathtt{R}_{DOM} \rightarrow \mathcal{L}^4$ maps each node reference to a tuple of four security levels. Hence, given a DOM reference r and a labelling Ξ, $\Xi(r) = \langle \sigma_n, \sigma_v, \sigma_p, \sigma_s \rangle$, where: **(1)** σ_n is the node level, **(2)** σ_v is the value level, **(3)** σ_p is the position level, and **(4)** σ_s is the structure security level. For clarity, given a node n pointed to by a reference r and a node labelling Ξ, we denote by $\Xi(r).\mathsf{node}$, $\Xi(r).\mathsf{value}$, $\Xi(r).\mathsf{pos}$, and $\Xi(r).\mathsf{struct}$ its node level, value level, position level, and structure security level, respectively. We impose four restrictions on the levels assigned to a given node.

1. One cannot store a visible value inside an invisible node. Formally, for every node reference $r \in dom(\Xi)$, it holds that: $\Xi(r).\mathsf{node} \sqsubseteq \Xi(r).\mathsf{value}$.

2. An invisible node cannot have a visible position. Formally, for every node reference $r \in dom(\Xi)$, it holds that: $\Xi(r).\mathsf{node} \sqsubseteq \Xi(r).\mathsf{pos}$.

3. An invisible node cannot have a visible number of children. Formally, for every node reference $r \in dom(\Xi)$, it holds that: $\Xi(r).\mathsf{node} \sqsubseteq \Xi(r).\mathsf{struct}$.

4. An invisible node cannot have a visible child. This means that programs are not allowed to add visible nodes to the list of children of invisible nodes. Formally, for every two node references $r, r' \in dom(\Xi)$ such that $f(r).\mathsf{children}(i) = r'$ for some integer i, it holds that $\Xi(r).\mathsf{node} \sqsubseteq \Xi(r').\mathsf{node}$.

Finally, as discussed in Section 7.2, the position levels of the children of labelled nodes must be monotonically increasing. All the constraints that DOM labellings must verify (and which are enforced by the monitor plugins introduced in the following subsection) are formally presented in Definition 7.2.

Definition 7.2 (Well-Labelled Forest)**.** *A Core DOM forest $f \in \mathtt{F}_{DOM}$ is said to be well-labelled by a labelling $\Xi \in \mathtt{Lab}_{DOM}$, if for every reference $r \in dom(\Xi)$, it holds that:*

- $\Xi(r).\mathsf{node} \sqsubseteq \Xi(r).\mathsf{value} \sqcap \Xi(r).\mathsf{pos} \sqcap \Xi(r).\mathsf{struct}$,
- $\forall_{0 \leq i < |f(r).\mathsf{children}|}\ \Xi(r).\mathsf{node} \sqsubseteq \Xi(f(r).\mathsf{children}(i)).\mathsf{node}$,
- $\forall_{0 \leq i < j < |f(r).\mathsf{children}|}\ \Xi(f(r).\mathsf{children}(i)).\mathsf{pos} \sqsubseteq \Xi(f(r).\mathsf{children}(j)).\mathsf{pos}$.

Informally, given a forest f labelled by Ξ, an attacker at level σ can see:

- the existence of nodes with observable node levels,
- the value stored in nodes with observable value levels,
- the position of nodes with observable position levels, and
- the number of children of nodes with observable structure security levels.

Furthermore, if a node is observable, then its tag, node level, position level, and structure security level are also assumed to be observable. The low-projection of a Core DOM forest $f \in \mathtt{F}_{DOM}$ labelled by $\Xi \in \mathtt{Lab}_{DOM}$ at a given security level σ is formally given in Definition 7.3.

Definition 7.3 (Low-Projection and Low-Equality for Core DOM forests)**.** *The low-projection of a forest $f \in \mathtt{F}_{DOM}$ w.r.t. a security level σ and a labelling $\Xi \in \mathtt{Lab}_{DOM}$ is given by:*

$$
\begin{aligned}
f \upharpoonright^{\Xi,\sigma} = \ & \{(r, f(r).\mathsf{tag}, \Xi(r).\mathsf{node}, \Xi(r).\mathsf{pos}, \Xi(r).\mathsf{struct}) \mid \Xi(r).\mathsf{node} \sqsubseteq \sigma\} \\
\cup\ & \{(r, f(r).\mathsf{value}, \Xi(r).\mathsf{value}) \mid\ \Xi(r).\mathsf{value} \sqsubseteq \sigma\} \\
\cup\ & \{(r, i, r') \mid\ f(r).\mathsf{children}(i) = r' \ \wedge\ \Xi(r').\mathsf{pos} \sqsubseteq \sigma\} \\
\cup\ & \{(r, \mathsf{null}) \mid\ f(r).\mathsf{parent} = \mathsf{null} \ \wedge\ \Xi(r).\mathsf{pos} \sqsubseteq \sigma\} \\
\cup\ & \{(r, |f(r).\mathsf{children}|) \mid\ \Xi(r).\mathsf{struct} \sqsubseteq \sigma\}
\end{aligned}
$$

Two forests f_0 and f_1, respectively labelled by Ξ_0 and Ξ_1 are said to be low-equal at security level σ, written $f_0, \Xi_0 \sim^{\sigma}_{DOM} f_1, \Xi_1$, if they coincide in their respective low-projections, meaning that $f_0 \upharpoonright^{\Xi_0,\sigma} = f_1 \upharpoonright^{\Xi_1,\sigma}$.

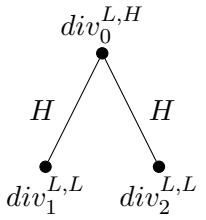
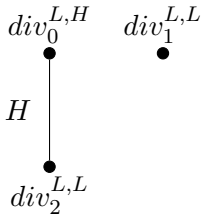
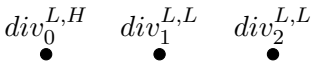
Final Forest		*Final Forest Low-Projection*
`h` = 0	`h` = 1	Both `h` = 0 and `h` = 1
		

Table 7.3: Two Core DOM forests and Their Low-Projections

Table 7.3 illustrates the final forests obtained from the execution of the Program 7.4 in two distinct memories that initially map the *high* variable `h` to 0 and to 1, respectively. On the left side of the table, we represent the two the final forests. And, on the right side of the table, we represent their coinciding low-projections. The position levels of the nodes bound to `div1` and `div2` as well as the structure security level of the node bound to `div0` are assumed to be originally set to H (*high*). All the other labels are assumed to be originally set to L (*low*). Each node is labelled with its corresponding node level and structure security level. Each edge connects a node (represented above) to one of its child nodes (represented below). The edge is labelled with the position level of the corresponding child node. Siblings are represented from left to right. Concretely, if n_1 and n_2 are siblings and the index of n_1 is lower than the index of n_2, then n_1 is represented on the left of n_2.

7.2.3 Monitor Plugins for the Core DOM API

As discussed in Chapter 6, a formal monitored API is modelled as a tuple $\langle \mathcal{S}, \mathcal{S}_{lab}, \mathcal{P}, \mathcal{P}_{lab}, \mathcal{R}_{IF}, \sim_{api} \rangle$ consisting of a set of API states, a set of API labellings, a set of plugins that operate on the API states, a set of monitor plugins that operate on the monitor labellings, a monitored API register, and a low-equality relation between labelled API states. Hence, the monitored Core DOM API is formally modelled as the tuple:

$$\langle \mathtt{F}_{DOM}, \mathtt{Lab}_{DOM}, \mathcal{P}^{DOM}, \mathcal{P}^{DOM}_{lab}, \mathcal{R}^{DOM}_{IF}, \sim_{DOM} \rangle \tag{7.5}$$

In the following, we use $\mathsf{CoreDOM}_{IF}$ to refer to the monitored Core DOM API. The only element of the monitored Core DOM API model that remains to be defined is the set $\mathcal{P}^{DOM}$ of monitor plugins.

The formal specification of the **monitor plugins** in $\mathcal{P}^{DOM}_{lab}$ is presented in Figure 7.3. As described in Chapter 6, each monitor plugin $\mathsf{dplug}_{lab} \in \mathcal{P}^{DOM}_{lab}$ is modelled as a relation of the form $\langle \Xi, \overrightarrow{\sigma} \rangle^{\beta}\ \mathsf{dplug}_{lab}\ \langle \Xi', \sigma \rangle$ where: **(1)** Ξ and Ξ' are the DOM labellings immediately before and after the execution of the plugin, **(2)** $\overrightarrow{\sigma}$ the sequence of levels that label the arguments given to the plugin, **(3)** β an internal event generated by the plugin to provide additional information to the monitor plugin, and **(4)** σ the level of the return value of the plugin – called the *reading effect* of the plugin.

Before proceeding to the description the monitor plugins, it is important to recall that the choice of which plugin to apply exclusively depends on the values of its first two arguments. Hence, in order to verify confinement (as defined in Definition 6.3) every monitor plugin must check whether the levels of the resources which it updates/creates are higher than or equal to the levels of the first two arguments. The monitor plugins of the monitored Core DOM API are briefly described below.

$$\textsc{New}\quad \frac{\sigma' = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \sigma_n \sqsubseteq \sigma_p \sqcap \sigma_s \qquad \Xi' = \Xi\,[r \mapsto \langle \sigma_n, \sigma_n, \sigma_p, \sigma_s \rangle]}{\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^{(r,\sigma_n,\sigma_p,\sigma_s)}\ \mathsf{new}_{lab}\ \langle \Xi', \sigma' \rangle}$$

$$\textsc{Append}\quad \frac{\begin{array}{c} r'' = \mathsf{null} \vee \Xi(r'').\mathsf{pos} \sqsubseteq \Xi(r').\mathsf{pos} \qquad \Xi(r).\mathsf{node} \sqsubseteq \Xi(r').\mathsf{node} \\ \sigma' = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \Xi(r).\mathsf{struct} \sqcap \Xi(r').\mathsf{pos} \end{array}}{\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^{(r,r',r'')}\ \mathsf{append}_{lab}\ \langle \Xi, \sigma' \rangle}$$

$$\textsc{Remove}\quad \frac{\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \Xi(r).\mathsf{struct} \sqcap \Xi(r').\mathsf{pos}}{\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^{(r,r')}\ \mathsf{remove}_{lab}\ \langle \Xi, \Xi(r').\mathsf{pos} \rangle}$$

$$\textsc{Item}\quad \frac{\sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Xi(r).\mathsf{pos}}{\langle \Xi, \sigma_0 :: \sigma_1 \rangle^{(r)}\ \mathsf{item}_{lab}\ \langle \Xi, \sigma \rangle}$$

$$\textsc{Length}\quad \frac{\sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Xi(r).\mathsf{struct}}{\langle \Xi, \sigma_0 :: \sigma_1 \rangle^{(r)}\ \mathsf{length}_{lab}\ \langle \Xi, \sigma \rangle}$$

$$\textsc{Parent}\quad \frac{\sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Xi(r).\mathsf{pos}}{\langle \Xi, \sigma_0 :: \sigma_1 \rangle^{(r)}\ \mathsf{parent}_{lab}\ \langle \Xi, \sigma \rangle}$$

$$\textsc{Value}\quad \frac{\sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Xi(r).\mathsf{value}}{\langle \Xi, \sigma_0 :: \sigma_1 \rangle^{(r)}\ \mathsf{value}_{lab}\ \langle \Xi, \sigma \rangle}$$

$$\textsc{Store}\quad \frac{\begin{array}{c} \sigma = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqcup \Xi(r).\mathsf{node} \qquad \sigma_0 \sqcup \sigma_1 \sqsubseteq \Xi(r).\mathsf{value} \\ \Xi' = \Xi\,[r \mapsto \langle \Xi(r).\mathsf{node}, \sigma, \Xi(r).\mathsf{pos}, \Xi(r).\mathsf{struct} \rangle] \end{array}}{\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^{(r)}\ \mathsf{store}_{lab}\ \langle \Xi', \sigma \rangle}$$

$$\textsc{Core DOM Monitor Plugins}\quad \mathcal{P}^{DOM}_{lab} = \{\ \mathsf{new}_{lab}, \mathsf{append}_{lab}, \mathsf{remove}_{lab}, \mathsf{item}_{lab}, \mathsf{length}_{lab}, \mathsf{parent}_{lab}, \mathsf{value}_{lab}, \mathsf{store}_{lab}\ \}$$

Figure 7.3: Core DOM Monitor - Primitives for Tree Operations

- [New] The internal event given to this monitor plugin is a 4-tuple that contains: the reference r in which the new Core DOM node is to be allocated as well as its node level σ_n, position level σ_p, and structure security level σ_s. While r is dynamically created by new, the three levels σ_n, σ_p, and σ_s are supposed to annotate the method call that triggered the invocation of the plugin. The extensible Core JavaScript Monitor "transmits" this annotation to new, which in turn "re-transmits" it to new_{lab} using the internal event mechanism. (Observe that in the specification of new, the DOM allocator is given σ_n as an argument. This means, as described in Section 3.3, that the node allocation takes place at level σ_n).

  The monitor plugin first checks whether the node level of the node to be created is higher than or equal to *lub* between the levels of the arguments of the plugin. This constraint prevents both the creation of a visible node depending on secret information and the creation of a visible node using an invisible tag name. Then, the monitor plugin checks whether the node level of the newly created node is lower than or equal to its structure security level and position level. This constraint prevents the creation of a node that does not verify the well-labelling predicate (established in Definition 7.2). Finally, the reading effect of the API method call is the node level of the newly created node.

- [Append] The internal event given to this monitor plugin is a 3-tuple that contains: **(1)** the reference r of the node to which a new node is to be appended, **(2)** the reference r' of the node to be appended, and **(3)** the reference of the current last child of the node pointed

to by r (which is to be the new left sibling of the node pointed to by r'). The monitor plugin first checks whether the structure security level of the node pointed to by r and the position level of the node pointed to by r' are greater than or equal to the *lub* between the levels of the arguments of the plugin. This constraint prevents **(1)** the insertion of a node with a visible position depending on secret information and **(2)** the changing of the number of children of a node with a visible number of children depending on secret information. When $r'' \neq \mathsf{null}$, this monitor plugin also checks whether the position level of the node pointed to by r' is higher than or equal to the position level of the node pointed to by r'', which is to be its new left-sibling. This constraint ensures that the position levels of the children of every Core DOM node are always monotonically increasing. The reading effect of the plugin is the *lub* between the levels of its arguments.

- [Remove] The internal event given to this monitor plugin is a 2-tuple that contains: **(1)** the reference r of the node from which a node is to be removed and **(2)** the reference r' of the node to be removed. The monitor plugin first checks whether the structure security level of the node pointed to by r and the position level of the node pointed to by r' are greater than or equal to the *lub* between the levels of the arguments of the plugin. This constraint prevents **(1)** the removal of a node with a visible position depending on secret information and **(2)** the changing of the number of children of a node with a visible number of children depending on secret information. The reading effect of the plugin is the *lub* between the levels of its arguments.

- [Item] The internal event given to this monitor plugin is a 1-tuple that contains the reference r of the node that is to be obtained from the list of children of its parent. The reading effect of this plugin is the *lub* between the levels of its arguments and the **position level** of the node pointed to by r.

- [Length] The internal event given to this monitor plugin is a 1-tuple that contains the reference r of the node whose number of elements is being inspected. The reading effect of this plugin is the *lub* between the levels of its arguments and the **structure security level** of the node pointed to by r.

- [Parent] The internal event given to this monitor plugin is a 1-tuple that contains the reference r whose parent is to be inspected. The reading effect of this plugin is the *lub* between the levels of its arguments and the **position level** of the node pointed to by r.

- [Value] The internal event given to this monitor plugin is a 1-tuple that contains the reference r of the node whose value is to be inspected. The reading effect of this plugin is the *lub* between the levels of its arguments and the **value level** of the node pointed to by r.

- [Store] The internal event given to this monitor plugin is a 1-tuple that contains the reference r pointing to the node in which a new value is to be stored. The monitor plugin first checks whether the value level of that node is higher than or equal to the *lub* between the levels of the first two arguments of the plugin. This constraint prevents **sensitive upgrades**. That is, it prevents the value level of a node with an observable value level from being upgraded within an unobservable context. The monitor plugin then updates the value level of the node pointed to by r with the *lub* between its node level and the levels of the arguments of the plugin. This level is also used as the reading effect of the plugin. Observe that this monitor plugin updates the API labelling in way that preserves the well-labelling predicate for nodes (since the new value level of the node pointed to by r is higher than or equal to its node level).

7.2.4 Soundness

This section presents the three main properties of the monitor extensions for the Core DOM API:

- Lemma 7.1 states that the monitored execution of every Core DOM plugin preserves the well-labelling predicate for Core DOM forests (Definition 7.2).
- Lemma 7.2 states that the monitored Core DOM API is confined according to Definition 6.3.
- Finally, Theorem 7.1 states that the monitored Core DOM API is noninterferent according to Definition 6.4.

The proofs of the results can be found in **Appendix D.1**.

Lemma 7.1 (Well-labelling Preservation)**.** *For any Core DOM monitored plugin* $(\mathsf{dplug}, \mathsf{dplug}_{lab}) \in rng(\mathcal{R}_{IF}^{DOM})$*, forests* $f_0, f_1 \in \mathtt{F}_{DOM}$*, labellings* $\Xi_0, \Xi_1 \in \mathtt{Lab}_{DOM}$*, annotation* α*, sequence of values* $\overrightarrow{v}$*, sequence of levels* $\overrightarrow{\sigma}$*, value* v*, level* σ*, and internal event* β*, such that:*

- f_0 *is well-labelled by* Ξ_0*,*
- $\langle f_0, \overrightarrow{v} \rangle^{\alpha}$ dplug $\langle f_1, v \rangle^{\beta}$*, and*
- $\langle \Xi_0, \overrightarrow{\sigma} \rangle^{\beta}$ dplug_{lab} $\langle \Xi_1, \sigma \rangle$

It holds that: f_1 *is well-labelled by* Ξ_1*.*

Lemma 7.2 (Confinement of the Monitored Core DOM API)**.** *The API* $\mathsf{CoreDOM}_{IF}$ *is confined.*

Theorem 7.1 (Noninterference of the Monitored Core DOM API)**.** $\mathbf{NI}(\mathsf{CoreDOM}_{IF})$.

An immediate corollary of Theorems 7.1 and 6.1 is that the plugging of the Core DOM API into the extensible Core JavaScript monitor yields a noninterferent extended monitor.

Corollary 7.1 (Noninterference - (Core JavaScript + Core DOM) Monitor)**.** $\mathbf{NI}(\Downarrow_{IF}^{\mathsf{CoreDOM}_{IF}})$.

7.3 Secure Information Flow for Live Collections

Live collections are a special type of data structure featured in the DOM API that automatically and dynamically reflect modifications to the document. The DOM API includes several methods that return live collections. For instance, the method `getElementsByTagName` returns a live collection containing all the nodes in the document tree whose tag matches the string given as input. Since a live collection automatically reflects modifications to the document, every time a node matching the query that generated a given live collection is inserted/removed in/from the document, it is also automatically inserted/removed in/from that live collection. Therefore, rather than a simple static data structure, a live collection is in fact a dynamic query to the document.

The nodes of a live collection are arranged in *document order*. According to the specification, the order of a node is determined by the position in which "the first character of [its] XML representation occurs in the XML representation of the document after expansion of general entities" [W3C Recommendation 2005]. In other words, the document order is an ordering $\leq$ on the nodes of the Core DOM forest such that for every two nodes n_0 and n_1 in the same DOM

$r \in \mathtt{R}_{DOM} \subset \mathtt{Ref}$			% DOM References
$r_0, \cdots, r_i \in \mathtt{R}_{\lightning} \subset \mathtt{Ref}$			% Live References
$ln \in \mathtt{Node}_{\lightning}$	$::=$	$\langle r, m \rangle$	% Live Node
$lives \in \mathtt{Lives}$	$::=$	$[r_0 \mapsto ln_0, \cdots, r_i \mapsto ln_i]$	% Live Record
$\nu \in \mathtt{F}_{\lightning}$	$::=$	$\langle f, lives \rangle$	% DOM State
$\mathsf{dplug} \in \mathcal{P}^{\lightning}$			% Core DOM Plugins + Live Collections
$\mathcal{R}^{\lightning} : \mathtt{Ref} \times \mathtt{Prim} \to \mathcal{P}^{\lightning}$			% Core DOM Register
$\mathsf{CoreDOM}^{\lightning} \in \mathcal{A}$		$\langle \mathtt{F}_{DOM}, \mathcal{P}^{\lightning}, \mathcal{R}^{\lightning} \rangle$	% Core DOM API

Table 7.4: DOM - Semantic Domains

tree, $n_0 \leq n_1$ if and only if n_0 is found before n_1 in a **depth-first left-to-right** search starting from the root of that tree.

In this section, we extend the Core DOM API with the following methods and properties for handling live collections:

- `node.getElementsByTagName(tag)`: creates a new live collection containing all the descendants of `node` with tag `tag` in document order;
- `lc[i]`: retrieves the i+1$^{\text{th}}$ node in the live collection bound to `lc`;
- `lc.length`: returns the number of nodes in the live collection bound to `lc`.

7.3.1 Extending the Formal DOM API with Live Collections

Live collections are not part of the DOM forest, but a different type of data structure. Hence, we consider a new type of node, called a *live collection node*, taken from a set $\mathtt{Node}_{\lightning}$, which are used to model the live collections created during the execution. Consequently, the set of Core DOM states must be redefined so that Core DOM states may include live collections. Hence, a Core DOM API state $\nu \in \mathtt{F}_{\lightning}$ is now modelled as a pair $\langle f, lives \rangle$ consisting of a Core DOM forest $f \in \mathtt{F}_{DOM}$ and a partial function $lives \in \mathtt{Lives} : \mathtt{R}_{\lightning} \to \mathtt{Node}_{\lightning}$, which we call *live collection register*. A live collection register maps live collection references in a set $\mathtt{R}_{\lightning}$ to live collection nodes. For clarity, given a DOM API state ν, we denote by $\nu.f$ and $\nu.lives$ its corresponding Core DOM forest and live collection register. Furthermore, the elements of $\mathtt{F}_{DOM}$ are called *tree nodes*, whereas the elements of $\mathtt{Node}_{\lightning}$ are called *live collection nodes*.

The Core DOM API extended with live collections is formally modelled as the triple $\langle \mathtt{F}_{\lightning}, \mathcal{P}^{\lightning}, \mathcal{R}^{\lightning} \rangle$, where $\mathcal{P}^{\lightning}$ is the set of the Core DOM plugins extended with the plugins required for the manipulation of live collections and $\mathcal{R}^{\lightning} : \mathtt{Ref} \times \mathtt{Prim} \to \mathcal{P}^{\lightning}$ is the extension of the Core DOM register that includes those additional plugins. In the following, we use $\mathsf{CoreDOM}^{\lightning}$ to refer to the Core DOM API extended with live collections. The new components of the Core DOM API formal model extended with live collections are summarised in Table 7.4.

In modelling the semantics of live collections, we chose to re-compute the content of a live collection every time a program tries to look-up one of its elements or its number of elements. The alternative approach would be to compute it only once and, every time there were changes in the document's structure, to reflect those changes in all existing live collections. This second approach has the disadvantage of scattering the semantics of live collections through all the Core DOM plugins that modify the structure of the document.

$$\mathcal{R}^{\lambda}_{IF}(r_0, v_1) = \begin{cases} (\mathsf{new}_{\lambda}, \mathsf{new}^{\lambda}_{lab}) & \text{if } r_0 \in \mathtt{R}_{DOM} \ \wedge \ v_1 = \texttt{"getElementsByTagName"} \\ (\mathsf{item}_{\lambda}, \mathsf{item}^{\lambda}_{lab}) & \text{if } r_0 \in \mathtt{R}_{\lambda} \ \wedge \ v_1 \in \mathtt{Num} \\ (\mathsf{length}_{\lambda}, \mathsf{length}^{\lambda}_{lab}) & \text{if } r_0 \in \mathtt{R}_{\lambda} \ \wedge \ v_1 = \texttt{"length"} \\ (\mathsf{redirect}_{\lambda}, \mathsf{redirect}^{\lambda}_{lab}) & \text{if } (r_0, v_1) \in dom(\mathcal{R}^{DOM}_{IF}) \end{cases}$$

Figure 7.4: The Live Collection API Register: $\mathcal{R}^{\lambda}$

$$\frac{|f(r).\mathsf{children}| = 0 \qquad f(r).\mathsf{tag} \neq m}{f \vdash r \rightsquigarrow_m \varepsilon}$$ Node Not Found - Leaf Node

$$\frac{\overrightarrow{r} = f(r).\mathsf{children} \qquad |\overrightarrow{r}| = n \qquad f(r).\mathsf{tag} \neq m \qquad \forall_{0 \leq i < n} \ f \vdash \overrightarrow{r}(i) \rightsquigarrow_m \overrightarrow{r}_i}{f \vdash r \rightsquigarrow_m \overrightarrow{r}_0 :: \cdots :: \overrightarrow{r}_{n-1}}$$ Node Not Found - Non-Leaf Node

$$\frac{|f(r).\mathsf{children}| = 0 \qquad f(r).\mathsf{tag} = m}{f \vdash r \rightsquigarrow_m r :: \varepsilon}$$ Node Found - Leaf Node

$$\frac{\overrightarrow{r} = f(r).\mathsf{children} \qquad |\overrightarrow{r}| = n \qquad f(r).\mathsf{tag} = m \qquad \forall_{0 \leq i < n} \ f \vdash \overrightarrow{r}(i) \rightsquigarrow_m \overrightarrow{r}_i}{f \vdash r \rightsquigarrow_m r :: \overrightarrow{r}_0 :: \cdots :: \overrightarrow{r}_{n-1}}$$ Node Found - Non-Leaf Node

Figure 7.5: Search Predicate

Formally, a *live collection node* is modelled as a tuple of the form $\langle r, m \rangle$, where r is the reference of the DOM node on which the query was issued and m the corresponding query. For instance, the evaluation of `div0.getElementsByTagName("DIV")` generates a live collection that contains the reference of the node bound to `div0` and the string `"DIV"`. Analogously to tree nodes, live collections are allocated in a set of references, that does not overlap with either the one used for the allocation of Core JavaScript objects or the one used for the allocation of tree nodes. Consequently, the allocator of live collections $\mathsf{fresh}_{live} : \mathcal{L} \to \mathtt{R}_{\lambda}$ is assumed to generate references in a set that does not overlap with the sets used for the allocation of Core DOM nodes and Core JavaScript objects.

In order to avoid repetition, we omit the specification of the API register $\mathcal{R}^{\lambda}$. Instead, Figure 7.4 presents its monitored version $\mathcal{R}^{\lambda}_{IF} : \mathtt{Ref} \times \mathtt{Prim} \to \mathcal{P}^{\lambda} \times \mathcal{P}^{\lambda}_{lab}$.[2] The conditions under which each plugin for handling live collections is executed are explained below.

- When a program invokes a method named `"getElementsByTagName"` on a tree node, the plugin new_{λ} is executed.
- When a program inspects an integer property of a live collection node, the plugin item_{λ} is executed.
- When a program inspects the property `"length"` of a live collection node, the plugin $\mathsf{length}_{\lambda}$ is executed.
- Finally, every expression intercepted by the API register of the Core DOM monitor without live collections is also intercepted by its extension with live collections. In those cases, the plugin $\mathsf{redirect}_{\lambda}$ redirects the call to the appropriate Core DOM API plugin.

The semantics of live collections makes use of a search predicate of the form $f \vdash r \rightsquigarrow_m \overrightarrow{r}$, formally given in Figure 7.5. This predicate formalises the search for the nodes matching a given

[2] As mentioned before, to the obtain the unmonitored register from its monitored version, it suffices to ignore the second component of the output.

$$\textsc{Live New}\quad \frac{r' = \mathsf{fresh}_{live}(\sigma_l) \qquad lives' = \nu.lives\,[r' \mapsto \langle r, m\rangle]}{\langle \nu, r :: \texttt{"getElementByTagName"} :: m\rangle^{\sigma_l}\ \mathsf{new}_{\not\iota}\ \langle\langle \nu.f, lives'\rangle, r'\rangle^{(r',\sigma_l)}}$$

$$\textsc{Live Item}\quad \frac{\nu.lives(r) = \langle r', m\rangle \qquad \nu.f \vdash r' \rightsquigarrow_m \overrightarrow{r} \qquad \overrightarrow{r}(i) = r''}{\langle \nu, r :: i\rangle\ \mathsf{item}_{\not\iota}\ \langle \nu, r''\rangle^{(\nu.f,r,r',r'')}}$$

$$\textsc{Live Length}\quad \frac{\nu.lives(r) = \langle r', m\rangle \qquad \nu.f \vdash r' \rightsquigarrow_m \overrightarrow{r}}{\langle \nu, r :: \texttt{"length"}\rangle\ \mathsf{length}_{\not\iota}\ \langle \nu, |\overrightarrow{r}|\rangle^{(\nu.f,r,r',m)}}$$

$$\textsc{Core DOM Redirection}\quad \frac{\mathsf{dplug} = \mathcal{R}^{DOM}(\overrightarrow{v}(0), \overrightarrow{v}(1)) \qquad \langle \nu.f, \overrightarrow{v}\rangle\ \mathsf{dplug}\ \langle f', v\rangle^{\beta}}{\langle \nu, \overrightarrow{v}\rangle\ \mathsf{redirect}_{\not\iota}\ \langle\langle f', \nu.lives\rangle, v\rangle^{\beta}}$$

$$\textsc{Core DOM + Live Collections Plugins}\quad \mathcal{P}^{\not\iota} = \{\ \mathsf{new}_{\not\iota}, \mathsf{item}_{\not\iota}, \mathsf{length}_{\not\iota}, \mathsf{redirect}_{\not\iota}\ \}$$

Figure 7.6: Core DOM API + Live Collections Plugins

tag in a DOM tree **in document order**. Intuitively, given a forest f, a node reference r, a tag name m, and a list of node references $\overrightarrow{r}$, $f \vdash r \rightsquigarrow_m \overrightarrow{r}$ means that $\overrightarrow{r}$ is the sequence that contains all the nodes with tag m found when traversing the tree of f rooted at r in **document order**. The plugins for handling live collections are formally presented in Figure 7.6 and are briefly described below.

- The plugin $\mathsf{new}_{\not\iota}$ expects as arguments: **(1)** the reference r of the node on which the live collection query was issued, **(2)** the string `"getElementByTagName"`, and **(3)** the tag name m of the nodes that will form the live collection. It then creates the live collection node $\langle r, m\rangle$ and allocates it in a new reference r' (computed using fresh_{live}). Finally, the plugin return value is the reference of the newly created live collection node.

- The plugin $\mathsf{item}_{\not\iota}$ expects arguments: **(1)** the reference r of the live collection node whose element is to be inspected, and **(2)** the position i that the element occupies in the live collection. It then inspects the live collection node, thereby obtaining the reference r' pointing to the Core DOM node on which the query was issued as well as the tag name m of the nodes that form the live collection. Using these two values, the rule computes a sequence $\overrightarrow{r}$ that contains the references of the descendants of the node pointed to by r' with tag name m in document order. Finally, the plugin return value is the $i + 1^{\text{th}}$ reference of $\overrightarrow{r}$.

- The plugin $\mathsf{length}_{\not\iota}$ expects as arguments: a node reference r and the string `"length"`. It proceeds as in the previous rule except that, instead of returning the $i + 1^{\text{th}}$ reference of $\overrightarrow{r}$, the return value of this plugin is the number of elements of $\overrightarrow{r}$.

- The plugin $\mathsf{redirect}_{\not\iota}$ is called every time a plugin in $\mathcal{P}^{DOM}$ is to be executed. Hence, the plugin $\mathsf{redirect}_{\not\iota}$ fetches from the Core DOM API register (without live collections) the Core DOM API plugin to execute. The second plugin is executed using as API state only the forest component of the current state.

7.3.2 Information Leaks introduced by Live Collections

Live collections can be exploited to encode new types of information leaks. We now discuss the main challenges imposed by the introduction of live collections as well as how we propose to tackle them.

7.3.2.1 Leaks via the Size of Live Collections

Consider the program below, which is to be executed in a forest that originally contains five orphan **DIV** nodes respectively bound to the variables `div0`, `div1`, `div2`, `div3`, and `div4`.

```
div0.appendChild(div1),
div0.appendChild(div2),
div0.appendChild(div3),
lc0 = div0.getElementsByTagName("DIV"),
h ? (div1.appendChild(div4)),
l0 = lc0.length
```
(7.6)

Depending on the initial value of the *high* variable `h`, the initially *low* variable `l0` is either either set to 4 or set to 5.

In order to tackle this type of leak, we require the programmer to pre-establish for each possible tag name m an upper bound on the position levels of the nodes with that tag name, which we denote by σ_m and call *global tag level*. For instance, $\sigma_{\mathbf{DIV}}$ corresponds to the pre-established upper bound on the position levels of **DIV** nodes. When the monitor evaluates the expression `lc0.length`, it first checks whether the position levels of all **DIV** nodes in the Core DOM forest are lower than or equal to the global position level. If that is the case, the execution is allowed to go through and the reading effect of the whole expression is $\sigma_{\mathbf{DIV}}$. Otherwise, the execution is aborted. Therefore, for this program to be legal the global tag level of **DIV** nodes $\sigma_{\mathbf{DIV}}$ must be set to H. Consequently, the reading effect of the whole expression is H.

The *global tag level* is used to control the implicit flows that can be encoded via the inspection of the number of elements of live collections. Hence, it cannot be flow-sensitive, since upgrading the global tag level constitutes a **sensitive upgrade**.

7.3.2.2 Order Leaks via the Inspection of Live Collections

The inspection of an element of a live collection leverages information about the position it occupies in that live collection and therefore in the document structure. Hence, live collections introduce a new type of *order leak*. Consider, for instance, the following program:

```
div0.appendChild(div1),
div0.appendChild(div2),
div0.appendChild(div3),
lc0 = div0.getElementsByTagName("DIV"),
h ? (div1.appendChild(div4)),
l0 = lc0[3]
```
(7.7)

Here, depending on the initial value of the *high* variable `h`, the initially low variable `l0` is assigned either to the node initially bound to `div3` or to the node initially bound to `div2`. Hence, the monitor must be able to detect that the evaluation of `lc0`[3] leaks information at level H.

Let us ignore by now the information flows triggered by the operations involving live collections during the execution of Program 7.7 and focus on the operations that only involve tree nodes. For the execution of this program to be legal (according to the current enforcement

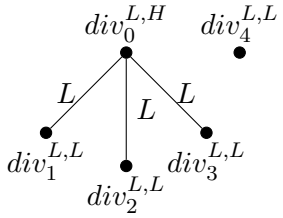
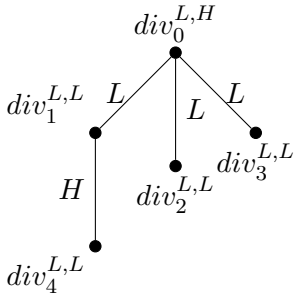
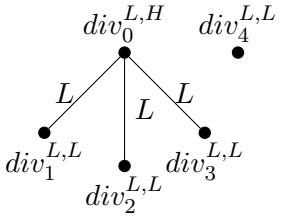
Final Forest		*Final Forest Low-Projection*
`h` = 0	`h` = 1	Both `h` = 0 and `h` = 1
		

Table 7.5: Two Core DOM forests and their Coinciding Low Projections

-

mechanism), the position level of the node bound to `div4` as well as the structure security level of the node bound to `div1` must be *high*. All other labels may be set to L. On its left side, Table 7.5 illustrates the final forests obtained from the execution of Program 7.7 in two distinct memories that initially map the `h` to 0 and to 1, respectively. On the right, it represents their (coinciding) low-projection. Although the two final forests are low-equal, the evaluation of the expression of `lc0`[3] in each of them yields two different values.

The example given above clearly shows that the use of live collections enhances the observational power of an attacker. This happens because live collections allow an attacker to operate on the nodes with the same tag in the same tree as if they were siblings. Hence, it is necessary to adjust the notion of a node's position in order to take into account this new way of traversing the DOM forest. Let the *live index* of a node in a given tree be its position in the list of nodes obtained by searching that tree for the nodes with its tag in document order. When a program uses live collections to traverse a given a tree, the position of every node in that tree must be understood as the triple consisting of its parent, its index, and its live index. Hence, changing the position of a node in a tree causes the positions of the nodes with the same tag with higher live indexes to change. In order to deal with this new type of order leak, the proposed enforcement mechanism guarantees that one can only inspect a live collection if the position levels of the nodes it "contains" monotonically increase in **document order**. For instance, in Table 7.5, when `h` is initially set to 1, the final tree rooted at the node bound to `div0` does not comply with this requirement. Concretely, while the position level of the node bound to `div4` is not lower than or equal to the position level of the node bound to `div2`, the live index of the node bound to `div4` is lower than the live index of that bound to `div2`.

7.3.3 An Attacker Model for Live Collections

At the formal level, the introduction of live collections poses an important challenge: how to model the enhanced observational power of an attacker that can use live collections to inspect the Core DOM forest? To answer this question formally means: **(1)** restating the low-equality definition for forests so as to correctly capture the observational power of such an attacker and **(2)** introducing a new low-equality for live collection registers. In order to do this, we have to extend the notion of DOM labelling to take into account live collection registers. Hence, an extended DOM labelling $\Xi \in \mathtt{Lab}_\ell$ is modelled as pair $\langle \Xi_0, \Xi_1 \rangle$, where $\Xi_0 \in \mathtt{Lab}_{DOM}$ is the *forest labelling* as defined in the previous section and $\Xi_1 \in: \mathtt{R}_\ell \to \mathcal{L}$ is the *live collection register labelling*. Informally, given a live collection reference $r \in \mathtt{R}_\ell$, $\Xi_1(r) = \sigma$ means that the existence

of the live collection pointed to by r is only visible at levels higher than or equal to σ. When a live collection is visible, the reference of the node in which the query was issued as well as the corresponding tag name are also visible. For simplicity, given a DOM labelling $\Xi = \langle \Xi_0, \Xi_1 \rangle$, Ξ_0 and Ξ_1 are respectively denoted by $\Xi.f$ and $\Xi.lives$.

The low-equality for live collection registers is given in Definition 7.4. As mentioned above, this definition simply states that an attacker at level σ can only see the existence of live collections labelled with levels $\sqsubseteq \sigma$.

Definition 7.4 (Low-Projection and Low-Equality for Live Collection Registers)**.** *The low-projection of a live collection register lives w.r.t. a security level σ and a live collection register labelling Ξ is given by:*

$$lives \upharpoonright_{\natural}^{\Xi,\sigma} = \{(r, r', m, \Xi(r)) \mid \ \Xi(r) \sqsubseteq \sigma \ \wedge \ lives(r) = \langle r', m \rangle\}$$

Two live collection registers $lives_0$ and $lives_1$, respectively labelled by Ξ_0 and Ξ_1, are said to be low-equal at security level σ, written $lives_0, \Xi_0 \sim_{\natural}^{\sigma} lives_1, \Xi_1$, if they coincide in their respective low-projections – $lives_0 \upharpoonright_{\natural}^{\Xi_0,\sigma} = lives_1 \upharpoonright_{\natural}^{\Xi_1,\sigma}$.

Besides giving a definition of low-equality for live collection registers, one must modify the definition of low-projection for Core DOM forests so that an attacker at level σ can additionally see: **(1)** the live indexes of the nodes whose position levels are $\sqsubseteq \sigma$ and **(2)** the number of descendants of visible nodes with a given tag whose global tag level is $\sqsubseteq \sigma$. Definiton 7.5 formally presents the new low-equality for Core DOM forests.

Definition 7.5 (Low-Projection and Low-Equality for Core DOM forests with Live Collections)**.** *The low-projection of a Core DOM forest f w.r.t. a security level σ and a labelling Ξ is given by:*

$$\begin{aligned} f \upharpoonright_{\natural}^{\Xi,\sigma} = \ & f \upharpoonright^{\Xi,\sigma} \\ & \cup \ \{(r, m, i, r') \mid \ f \vdash r \rightsquigarrow_m \overrightarrow{r'} \ \wedge \ \overrightarrow{r'}(i) = r' \ \wedge \ \Xi(r').\mathsf{pos} \sqsubseteq \sigma\} \\ & \cup \ \{(r, m, n) \mid \ f \vdash r \rightsquigarrow_m \overrightarrow{r'} \ \wedge \ |\overrightarrow{r'}| = n \ \wedge \ \sigma_m \sqcup \Xi(r).\mathsf{node} \sqsubseteq \sigma\} \end{aligned}$$

Two Core DOM forests ν_0 and ν_1, respectively labelled by Ξ_0 and Ξ_1, are said to be low-equal at security level σ, written $f_0, \Xi_0 \sim_{\natural}^{\sigma} f_1, \Xi_1$, if they coincide in their respective low-projections, meaning that $f_0 \upharpoonright_{\natural}^{\Xi_0,\sigma} = f_1 \upharpoonright_{\natural}^{\Xi_1,\sigma}$.

Finally, we define two different low-equality relations for labelled Core DOM API states.

- Two DOM states ν_0 and ν_1 respectively labelled by Ξ_0 and Ξ_1 are said to be low-equal at a given level σ if the corresponding forests are low-equal according to $\sim_{DOM}^{\sigma}$ and the corresponding live collection registers are low-equal according to $\sim_{\natural}^{\sigma}$:

$$\nu_0.f, \Xi_0.f \sim_{DOM}^{\sigma} \nu_1.f, \Xi_1.f \ \wedge \ \nu_0.lives, \Xi_0.lives \sim_{\natural}^{\sigma} \nu_1.lives, \Xi_1.lives$$

- Two DOM states ν_0 and ν_1 respectively labelled by Ξ_0 and Ξ_1 are low-equal *for live collections* at a given level σ if the corresponding forests are low-equal according to $\sim_{\natural}^{\sigma}$ (for forests) and the corresponding live collection registers are low-equal according to $\sim_{\natural}^{\sigma}$:

$$\nu_0.f, \Xi_0.f \sim_{\natural}^{\sigma} \nu_1.f, \Xi_1.f \ \wedge \ \nu_0.lives, \Xi_0.lives \sim_{\natural}^{\sigma} \nu_1.lives, \Xi_1.lives$$

7.3.3.1 Strengthening the Low-Equality for Core DOM forests

The new version of the low-equality for forests captures the additional power of an attacker who disposes of live collections to interact with the document. Hence, a possible way to proceed is to modify the previous monitor in order for it to enforce the stronger version of the low-equality. However, doing so would lead to stricter constraints regarding the way programs can modify the document, even if **no live collection is used to inspect its content**. Therefore, instead of imposing additional constraints on operations that update the content of the Core DOM forest, the new version of the monitor makes use of a predicate on Core DOM forests that checks **whether the inspection of the document via live collections is secure**. In a nutshell, any two labelled forests verifying this predicate and related by the first low-equality are also related by the new low-equality and, therefore, can be securely inspected using live collections. Informally, we say that a Core DOM forest f labelled by Ξ is *secure for live collections*, written $\mathcal{S}ec(f, \Xi)$, if:

- the position level of every node in f is lower than or equal to the global tag level corresponding to its tag,
- the position levels of the nodes with the same tag monotonically increase in document order,
- the position level of every node is lower than or equal to the position levels of all its descendants (this means that if the position of a node is secret, the positions of all its descendants are also secret).

The predicate $\mathcal{S}ec(f, \Xi)$ is defined with the help of a predicate $\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural \rightsquigarrow \phi'_\natural$, given in Definition 7.6, that holds if the tree rooted at r is *secure for live collections*.

Definition 7.6 (Secure Forest for Live Collections)**.** *The predicate* $\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural \rightsquigarrow \phi'_\natural$ *is recursively defined as follows:*

$$\textsc{Leaf Node}\quad \frac{\begin{array}{c} f(r).\mathsf{tag} = m \\ |f(r).\mathsf{children}| = 0 \\ \phi_\natural(m) \sqsubseteq \Xi(r).\mathsf{pos} \sqsubseteq \sigma_m \\ \phi'_\natural = \phi_\natural\,[m \mapsto \Xi(r).\mathsf{pos}] \end{array}}{\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural \rightsquigarrow \phi'_\natural}$$

$$\textsc{Non-Leaf Node}\quad \frac{\begin{array}{c} f(r).\mathsf{tag} = m \qquad \phi_\natural(m) \sqsubseteq \Xi(r).\mathsf{pos} \sqsubseteq \sigma_m \\ |f(r).\mathsf{children}| = n > 0 \qquad \phi^0_\natural = \phi_\natural\,[m \mapsto \Xi(r).\mathsf{pos}] \\ \forall_{0 \le i < n}\ \Xi(r).\mathsf{pos} \sqsubseteq \Xi(f(r).\mathsf{children}(i)).\mathsf{pos} \\ \forall_{0 \le i < n}\ \mathcal{S}ec_{f,\Xi} \vdash^{f(r).\mathsf{children}(i)} \phi^i_\natural \rightsquigarrow \phi^{i+1}_\natural \end{array}}{\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural \rightsquigarrow \phi^n_\natural}$$

In the definition above, the function $\phi_\natural$ maps each tag name to the position level of the last node with that tag name preceding the node pointed to by r in f in document order. The function $\phi'_\natural$ maps each tag name to the position level of the last node with that tag name in the tree rooted at r (if no such node exists, $\phi'_\natural$ coincides with $\phi_\natural$). Formally, the tree rooted at the node pointed to by r in Core DOM forest f labelled by Ξ is said to be well-labelled for live collections, written $\mathcal{S}ec(f, \Xi)$, if and only if there are two functions $\phi_\natural$ and $\phi'_\natural$ such that $\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural \rightsquigarrow \phi'_\natural$. A Core DOM forest is well-labelled for live collections if all its trees are well-labelled for live collections.

Theorem 7.2 (Low-Equality Strengthening)**.** *Given two forests* f_0 *and* f_1 *respectively labelled by* Ξ_0 *and* Ξ_1 *and a security level* σ *such that* $\mathcal{S}ec(f_0, \Xi_0)$ *and* $\mathcal{S}ec(f_1, \Xi_1)$ *and* $f_0, \Xi_0 \sim_\sigma f_1, \Xi_1$*, it holds that:* $f_0, \Xi_0 \sim^\sigma_\natural f_1, \Xi_1$*.*

The **proof** of Theorem 7.2 can be found in **Appendix D.2**.

$$\textsc{Live New}\quad \frac{\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \sigma \qquad \Xi' = \langle \Xi.f, \Xi.lives\,[r \mapsto \sigma] \rangle}{\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^{r,\sigma}\ \mathsf{new}^{\lightning}_{lab}\ \langle \Xi', \sigma \rangle}$$

$$\textsc{Live Length}\quad \frac{\sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Xi.lives(r) \sqcup \sigma_m \qquad \mathcal{S}ec(f, \Xi.f, r')}{\langle \Xi, \sigma_0 :: \sigma_1 \rangle^{(f,r,r',m)}\ \mathsf{length}^{\lightning}_{lab}\ \langle \Xi, \sigma \rangle}$$

$$\textsc{Live Item}\quad \frac{\sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Xi.lives(r) \sqcup \Xi.f(r'').\mathsf{pos} \qquad \mathcal{S}ec(f, \Xi.f, r')}{\langle \Xi, \sigma_0 :: \sigma_1 \rangle^{(f,r,r',r'')}\ \mathsf{item}^{\lightning}_{lab}\ \langle \Xi, \sigma \rangle}$$

$$\textsc{Core DOM Redirection}\quad \frac{(\mathsf{dplug}, \mathsf{dplug}_{lab}) = \mathcal{R}^{DOM}_{IF}(r_0, v_1) \qquad \langle \Xi.f, \overrightarrow{\sigma} \rangle^{\beta}\ \mathsf{dplug}_{lab}\ \langle \Xi', \sigma \rangle}{\langle \Xi, \overrightarrow{\sigma} \rangle^{(r_0, v_1, \beta)}\ \mathsf{redirect}^{\lightning}_{lab}\ \langle \langle \Xi', \Xi.lives \rangle, \sigma \rangle}$$

$$\textsc{Core DOM + Live Collections Monitor Plugins}\quad \mathcal{P}^{DOM}_{lab} = \{\ \mathsf{new}^{\lightning}_{lab}, \mathsf{length}^{\lightning}_{lab}, \mathsf{item}^{\lightning}_{lab}, \mathsf{redirect}^{\lightning}_{lab}\ \}$$

Figure 7.7: Core DOM Monitor - Live Collections

7.3.4 Monitor Plugins for the Core DOM API + Live Collections

The monitored Core DOM API extended with live collections is formally modelled as the tuple:

$$\langle \mathtt{F}_{\lightning}, \mathtt{Lab}_{\lightning}, \mathcal{P}^{\lightning}, \mathcal{P}^{\lightning}_{lab}, \mathcal{R}^{\lightning}_{IF}, \sim_{DOM} \rangle \tag{7.8}$$

In the following, we use $\mathsf{CoreDOM}^{\lightning}_{IF}$ to refer to the extension of the monitored Core DOM API with live collections. The only element of the monitored Core DOM API model that remains to be defined is the set $\mathcal{P}^{\lightning}_{lab}$ of monitor plugins. Observe that the low-equality relation to be used with $\mathsf{CoreDOM}^{\lightning}_{IF}$ is $\sim_{DOM}$ (and not $\sim_{\lightning}$). Hence, when a program interacts with live collections the corresponding monitor plugin verifies if the forest is well-labelled for live collections, in which case the execution is allowed to proceed. Otherwise, the execution is blocked. The monitor plugins for the Core DOM API extended with live collections are presented in Figure 7.7 and are briefly described below.

- [Live New] The internal event given to this monitor plugin consists of a 2-tuple containing: **(1)** the reference r of the newly allocated live collection node and **(2)** the levelσ of of the newly allocated live collection. This monitor plugin extends the current live collection register labelling with a mapping from r to σ. The monitor plugin checks whether the level of the new live collection node is higher than or equal to the the *lub* between the levels of all the arguments given to the plugin ($\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2$). This constraint prevents the creation of a visible live collection node depending on secret information and the creation of a visible live collection node associated with an invisible tag name. The reading effect of the plugin is the level of the newly allocated live collection node.

- [Live Length] The internal event given to this monitor plugin consists of a 4-tuple containing: **(1)** the current Core DOM forest f, **(2)** the reference r pointing to the live collection node whose number of elements is to be inspected, **(3)** the reference r' of the node in which the query was issued, and **(4)** the corresponding tag name m. This checks whether the tree rooted at r' is well-labelled for live collections. If it is the case, the reading effect of the plugin is the *lub* between: **(1)** the levels of the arguments (σ_0 and σ_1), **(2)** the global tag

level of m (σ_m), and **(3)** the level of the live collection node pointed to by r. Otherwise, the execution is blocked.

- [Live Item] The internal event given to this monitor plugin consists of a 4-tuple containing: **(1)** the current Core DOM forest f, **(2)** the reference r of the live collection node whose element is to be inspected, **(3)** the reference r' of the Core DOM node in which the query was issued, and **(4)** the reference r'' pointing to the Core DOM node to be returned by the plugin. This monitor plugin checks whether the tree rooted at r' is well-labelled for live collections. If it is the case, the reading effect of the plugin is the *lub* between: **(1)** the levels of the arguments (σ_0 and σ_1), **(2)** the position level of the node pointed to by r'', and **(3)** the level of the live collection node pointed to by r. Otherwise, the execution is blocked.

- [Redirect] The internal event given to this plugin consists of a 3-tuple containing: **(1)** the first argument r_0 of the call to the plugin , **(2)** the second argument v_1 of the call to the plugin, and **(3)** the internal event to be given to the corresponding Core DOM monitor plugin. The plugin uses the register of the monitored Core DOM API (not extended with live collections) to obtain its monitor plugin that is to be executed and executes it.

7.3.5 Soundness

This section presents the two main properties of the monitored version of the Core DOM API with live collections. Concretely:

- Lemma 7.3 states that the monitored Core DOM API is confined according to Definition 6.3.

- Finally, Theorem 7.3 states that the monitored Core DOM API is noninterferent according to Definition 6.4.

The proofs of the results can be found in **Appendix D.3**.

Lemma 7.3 (Confinement - Monitored Core DOM+ Live Collections)**.** *The API* $\mathsf{CoreDOM}^{\natural}_{IF}$ *is confined.*

Theorem 7.3 (Noninterference - Monitored Core DOM + Live Collections)**.** $\mathbf{NI}(\mathsf{CoreDOM}^{\natural}_{IF})$.

An immediate corollary of Theorems 7.3 and 6.1 is that the plugging of the monitored Core DOM API with live collections into the extensible Core JavaScript monitor yields a noninterferent extended monitor.

Corollary 7.2 (Noninterference - (Core JavaScript + Core DOM) Monitor)**.** $\mathbf{NI}(\Downarrow_{IF}^{\mathsf{CoreDOM}_{IF}})$.

7.4 Related Work

Secure Information Flow in Dynamic Tree Structures Russo et al. [Russo 2009] have been the first to study the problem of securing information flow in DOM-like dynamic tree structures. Their paper presents a monitor for a WHILE language with primitives for manipulating DOM-like trees as well as the corresponding proof of soundness. However, references are not modelled in this language. Instead, program configurations include the current working node of the program. This is, as the authors point out, the main difference between their model and JavaScript DOM operations (since in JavaScript, tree nodes are treated as first-class values).

By treating nodes as first-class values, we were able to give separate treatment to position leaks, which cannot be directly expressed in the language of [Russo 2009].[3]

The monitor presented in [Hedin 2014][4] includes a set of *statefull information-flow models* for tracking information flow in the DOM API including live collections. However, the authors only provide a general explanation of the techniques they use to control information flow in the DOM API and they do not prove any soundness property regarding these techniques. In particular, the paper includes an informal description of how to label and monitor the live collections returned by the method `getElementsByName`. In a nutshell, the proposed strategy consists of two steps. First, when a live collection is created, the node on which the query is issued is marked with the level of the newly created live collection. Second, the monitor restricts the operations that can be applied to that node and to its descendants. More concretely, an operation that may have an impact on the newly created live collection can only be performed in contexts lower than or equal to the level of that live collection. It is our opinion that this general approach can be used as an alternative to our enforcement mechanism. However, a detailed comparison between the two mechanisms would require having a detailed specification of the mechanism introduced in [Hedin 2014] as well as an argument justifying its soundness.

The authors of [Hedin 2012] include in their paper[5] a description of a JavaScript implementation of a DOM-like API. Hence, by using this DOM library implemented in JavaScript with the author's browser-instrumentation for enforcing information flow policies, it is possible to track secure information flow in JavaScript programs that interact with the DOM. Except that it is not the real DOM API, but a library that acts as the DOM implemented in JavaScript. The fact that the DOM API is neither implemented in JavaScript nor part of the JavaScript engine (interaction with the DOM is managed by a separate module of the browser [Grosskurth 2005]) requires the specification of monitor extensions the whole DOM API.

DOM Semantics Gardner et al. [Gardner 2008] proposed a compositional and concise formal specification of the DOM called Minimal DOM. The authors show that their semantics has no redundancy and that it is sufficient to describe the structural kernel of DOM Core Level 1. Informally, this means that the semantics of the un-modelled commands can be obtained from that of the modelled ones. Additionally, they apply local reasoning based on Separation Logic to prove invariant properties of JavaScript programs that interact with the DOM. Given that our aim is to track information flow in the DOM, we use a simplified semantics for DOM APIs that allows us to label DOM resources in a natural way. Like Minimal DOM, the Core DOM API is also compositional. Furthermore, all the primitives of Minimal DOM can be easily translated to the Core DOM API. Hence, we expect the authors' sufficiency claim to be applicable to Core DOM.

In his PhD thesis, Smith [Smith 2011] extended the fragment of the DOM API analysed in. [Gardner 2008] with a formalisation of all the *Fundamental Interfaces* of Core DOM Level 1 [W3C Recommendation 2005]. Hence, this formalisation includes live collections. In this formalisation, like in ours, live collections are lazy-evaluated. That is, the content of a live collection is recomputed every time that live collection is inspected. This approach to the modelling of live collections has the advantage of not scattering the semantics of live collections through the semantics of all the methods that interact with the DOM forest.

[3] Section 7.5 presents a detailed comparison between our model and the model of [Russo 2009].

[4] This work is also discussed in the Related Work Section of Chapter 6.

[5] This work, which resents a browser-instrumentation for enforcing information flow policies, is discussed in the Related Work Section of Chapter 4.

7.5 Discussion

7.5.1 Order Leaks in the DOM API

The DOM specification states that the children of a node constitute a collection of type `NodeList`. Every `NodeList` implements a method `item(index)` that "returns the indexth item in the collection" or `null` if the "index is greater than or equal to the number of nodes [it contains]" [W3C Recommendation 2005]. The Core DOM API allows the programmer to directly obtain the i^{th} child of a given node (like established in the DOM API), as well as to remove a node from an arbitrary position of the list of children of another node. This fact requires the enforcement mechanism to explicitly ensure that the position levels of sibling nodes are monotonically increasing. If we assume that every implementation of the DOM API forces a `NodeList` to be traversed from left to right, this problem automatically goes away due to standard label propagation. However, the specification makes no such restriction on the implementation of `NodeList`s and since such an implementation would be highly inefficient, it is reasonable to assume the opposite case.

7.5.2 A Comparison with the Model of Russo et al. [Russo 2009]

As we mentioned before, by modelling Core DOM nodes as first class values, we can naturally distinguish *order leaks* from *value leaks*. In other words, we can naturally distinguish the information flows regarding the position of a node from the information flows regarding the value which it stores. This distinction is not possible in the model of Russo et al. [Russo 2009] in that model the *position level* of a node coincides with its *node level*. In fact, in that model, it is not possible to change the position of a node in the DOM forest without deleting it and re-creating it – its position remains the same during its whole "lifetime". This makes it impossible to create a node with an invisible position that stores a visible value.

In order to better illustrate the point discussed above, we consider a concrete program that, when expressed in the model of [Russo 2009], causes the monitor to raise a security level of a resource that is not raised in our case.

```
n = document.createElement("DIV")^{L,H,L},
n.storeValue(l1),
h ? (
   document.appendChild(n),
   ⋮  // expensive computations
),
l2 = n.nodeValue
```
(7.9)

Suppose that this is program is executed in a labelled forest in which the structure security level of the *document* node is set to *high* (meaning that programs are allowed to append nodes to the root of the document inside *high* contexts). Program 7.9 proceeds as follows:

1. The program creates a **DIV** node with a *low* node level, a *high* position level, and a *low* structure security level,
2. The program assigns the node to variable `n`,
3. The program stores the value of the *low* variable `l1` inside the node,
4. If the value of the *high* variable `h` is not in `Falsy`, the program appends the node to list of children of the document node and then performs a series of expensive computations involving the document structure inside the *high* branch.

5. After executing the conditional, the program assigns the content of the node bound to `n` to the *low variable* `l2`.

This program cannot be equivalently expressed in the model of [Russo 2009], because in that model nodes must be created in the place they will occupy during the remaining of the execution. In this example, it means that the new node would have to be created inside the *high* conditional. Consequently, their monitor would need to upgrade the level of `l2` to *high*.

Chapter 8

Conclusions

Contents

In recent years, a lot of work has been dedicated to the study of information flow security in computing systems [Hedin 2011, Sabelfeld 2003a], with the double aim of preventing classified information from falling into the hands of unauthorised parties and preventing high-integrity resources from being updated depending on data coming from untrusted parties. However, it has been frequently observed that despite the *"ongoing attention from the research community, information-flow based enforcement mechanisms have not been widely (or even narrowly!) used"* [Zdancewic 2004]. Hence, the real challenge in Information Flow Control research is *"to find applications to all the existing results or, in failing to do so, provide a reasonable explanation for such failure"* [Zdancewic 2004]. This thesis tries to abridge this gap between theory and practice by studying a broad range of IFC mechanisms for a realistic core of a widely used programming language – JavaScript – which holds a prominent spot in the internet of today. Furthermore, we provide an implementation of the proposed mechanisms in order to illustrate how they can be used in practice.

In this final chapter we summarise the main technical contributions of this thesis, and give some perspective on future work.

8.1 Main Contributions

Hybrid Analysis While the dynamic features of JavaScript make it an exceedingly difficult target for static analysis [Maffeis 2009], dynamic methods for tracking information flow often impose a runtime overhead that is far from negligible [Hedin 2014]. Hence, we consider the hybrid type system presented in Chapter 5 a central contribution of this thesis, as it proposes a novel way to leverage the combination of runtime and static analyses in order to overcome some of the issues of these two approaches. We believe that this novel way of combining fully static type systems for checking secure information flow (such as those presented in [Volpano 1996] and [Banerjee 2002]) with program instrumentation can be replicated in other contexts for deriving more permissive hybrid mechanisms.

Extensible Security Monitors Besides the dynamicity of JavaScript, another important challenge to formal reasoning about client-side Web applications is the continuous emergence and heterogeneity of the APIs to which client-side scripts can resort while executing. To overcome this issue, we have presented an extensible security monitor for a core of JavaScript, which allows us to prove noninterference for Web APIs in a modular way and then plug the verified APIs into the extensible monitor in a way that preserves the security of the whole system. Furthermore, we have presented a general architecture for designing extensible monitor-inlining compilers so as to take practical advantage of the proposed mechanism for monitors.

Information Flow Analysis for the DOM API We have studied a set of monitor extensions to enforce secure information flow in a representative fragment of the Core DOM Level 1 API. The proposed solution tackles open issues in information flow security such as references and live collections in dynamic tree structures. By including references and live collections, the Core DOM API offers the expressive power of the real Core DOM Level 1 API in the form of a simple set of formal API specifications that is well tailored for automatic program analysis

8.2 Further Work

We envision the following tracks for future work:

- **Semantic Subtyping for Security Types.** Our subtyping relation for security types is very restrictive, since it requires the corresponding raw types to coincide. This may lead to the rejection of many secure programs. Hence, it would be interesting to use a more flexible notion of subtyping for security types with the proposed type systems, as that would render the two of them less restrictive. However, the design of such a notion of subtyping for security types is far from an easy challenge because of the use of recursion in the specification of security types.

  As in the works of Castagna et al. [Castagna 2005] in the context of safety types, a possible way to proceed is to ground the subtyping relation in a semantic criterion. For instance, by interpreting safety types as sets, one can use standard set-inclusion to define subtyping. Then, a safety type is a subtype of another if its denotational interpretation is contained in the denotational interpretation of the other. In fact, Sabelfeld and Sands [Sabelfeld 2001] already opened this way for information flow types with the study of an "extensional semantics-based formal specification of secure information-flow properties based on representing degrees of security by partial equivalence relations". In other words, in the authors' work, information flow types are interpreted as *partial equivalence relations (pers)* over a pre-established semantic domain. Then, an information flow type is less strict than another if the *per* of the former is contained in the *per* of the latter, meaning that stricter security types relate more "objects" in the semantic domain.

- **Hybrid Type Systems with Complex Assertions.** The hybrid type system we propose uses a simple program logic to reason about local scope. We conjecture that the use of a more expressive program logic (such as that of [Gardner 2012]) in the generation of the assertions to be verified at runtime would allow the hybrid mechanism to use **smaller** constraints. Therefore, it would have a positive impact on the performance of instrumented code and, consequently, on its applicability.

- **Automatic Synthesis of IFlow Signatures.** The development of sound IFlow Signatures for the secure extension of information flow monitors with new APIs is a technical undertaking that can be hardly left as task for the common programmer. Moreover, even if an API is implemented in JavaScript, its monitored execution is much more costly than the execution of its hypothetical IFlow Signature. Therefore, automatic synthesis of IFlow Signatures is an important issue to be considered in the design of extensible information flow monitors.

  However, it is worth noting that a precise analysis of this kind for fully-fledged JavaScript libraries is very unlikely to be attained because of the dynamic features of the language. Observe that such an analysis would even obviate the need for monitoring. Instead, one would just run the IFlow Signature of the whole program. Nevertheless, even if this type of analysis is not possible in general, the use of static flow-sensitive analyses, such as that

of [Hunt 2006], is a reasonable path to pursue for synthesising IFlow signatures of programs that do not take advantage of the most dynamic features of the language.

Bibliography

[3rd edition of ECMA 262 1999] The 3rd edition of ECMA 262. *ECMAScript Language Specification*. Rapport technique, ECMA, 1999. (Cited on pages 9, 10, 14 and 20.)

[5th edition of ECMA 262 2011] The 5th edition of ECMA 262. *ECMAScript Language Specification*. Rapport technique, ECMA, 2011. (Cited on pages 1, 9, 20, 21, 28 and 89.)

[Agat 2000] Johan Agat. *Transforming out Timing Leaks*. In Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '00, pages 40–53. ACM Press, 2000. (Cited on page 28.)

[Almeida Matos 2009] Ana Almeida Matos and Gérard Boudol. *On Declassification and the Non-Disclosure Policy*. volume 17, pages 549–597. IOS Press, 2009. (Cited on pages 3, 25 and 72.)

[Amadio 1991] Roberto M. Amadio and Luca Cardelli. *Subtyping the Recursive Types*. In Proceedings of the 18th ACM Symposium on Principles of Programming Languages, pages 104–118. ACM Press, 1991. (Cited on page 56.)

[Anderson 2005] Christopher Anderson, Paola Giannini and Sophia Drossopoulou. *Proceedings of the Towards Type Inference for JavaScript*. In 19th European Conference Object-Oriented Programming, Lecture Notes in Computer Science, pages 428–452. Springer, 2005. (Cited on pages 20 and 73.)

[Austin 2009] Thomas H. Austin and Cormac Flanagan. *Efficient Purely-Dynamic Information Flow Analysis*. In Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS '09, pages 113–124. ACM Press, 2009. (Cited on pages 32, 37, 47 and 86.)

[Austin 2010] Thomas H. Austin and Cormac Flanagan. *Permissive Dynamic Information Flow Analysis*. In Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS '10. ACM Press, 2010. (Cited on pages 37 and 47.)

[Austin 2012] Thomas H. Austin and Cormac Flanagan. *Multiple Facets for Dynamic Information Flow*. In Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, pages 165–178. ACM Press, 2012. (Cited on pages 31 and 47.)

[Banerjee 2002] Anindya Banerjee and David A. Naumann. *Secure Information Flow and Pointer Confinement in a Java-like Language*. In Proceedings of the 15th IEEE Computer Security Foundations Workshop, CSF'15, pages 253–267. IEEE Computer Society, 2002. (Cited on pages 3, 15, 28, 72 and 117.)

[Barth 2009] Adam Barth, Collin Jackson and John C. Mitchell. *Securing Frame Communications in Browsers*. Commun. ACM, vol. 52, no. 6, pages 83–91, 2009. (Cited on page 2.)

[Barth 2011] A. Barth. *The web origin concept*. In IETF, 2011. (Cited on page 1.)

[Bell 1976] David Elliott Bell and Leonard J. LaPadula. *Secure Computer Systems: Mathematical Foundations*. Rapport technique, Mitre Corp. Rep. MTR-2997 Rev. 1, 1976. (Cited on page 28.)

[Biba 1977] J. K. Biba. *Integrity Considerations for Secure Computer Systems*, 1977. (Cited on page 3.)

[Bielova 2011] Nataliia Bielova, Dominique Devriese, Fabio Massacci and Frank Piessens. *Reactive non-interference for a browser model.* In Proceedings of the 5th International Conference on Network and System Security, NSS'11, pages 97–104. IEEE Computer Society, 2011. (Cited on page 48.)

[Birgisson 2012] Arnar Birgisson, Daniel Hedin and Andrei Sabelfeld. *Boosting the Permissiveness of Dynamic Information-Flow Tracking by Testing.* In Proceedings of 19th European Symposium on Research in Computer Security, Lecture Notes in Computer Science, pages 55–72. Springer, 2012. (Cited on page 48.)

[Bodin 2013] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt and Gareth Smith. *A Trusted Mechanised JavaScript Specification.* In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, pages 87–100. ACM Press, 2013. (Cited on page 20.)

[Bohannon 2009] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich and Steve Zdancewic. *Reactive noninterference.* In Proceedings of the 16th ACM Conference on Computer and Communications Security, pages 79–90. ACM Press, 2009. (Cited on page 48.)

[Buiras 2014] Pablo Buiras, Deian Stefan and Alejandro Russo. *On Dynamic Flow-sensitive Floating Label Systems.* In Proceedings of the 27th IEEE Computer Security Foundations Symposium, CSF'27. IEEE Computer Society, 2014. (Cited on page 48.)

[Castagna 2005] Giuseppe Castagna and Alain Frisch. *A Gentle Introduction to Semantic Subtyping.* In Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP), volume 3580 of *Lecture Notes in Computer Science*, pages 30–34. Springer, 2005. (Cited on page 118.)

[Charguéraud 2013] Arthur Charguéraud. *Pretty-Big-Step Semantics.* In Programming Languages and Systems, volume 7792 of *Lecture Notes in Computer Science*, pages 41–60. Springer Berlin Heidelberg, 2013. (Cited on page 20.)

[Chudnov 2010] Andrey Chudnov and David A. Naumann. *Information Flow Monitor Inlining.* In Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF'10, pages 200–214. IEEE Computer Society, 2010. (Cited on pages 31, 43, 48 and 49.)

[Clements 2008] John Clements, Ayswarya Sundaram and David Herman. *Implementing continuation marks in JavaScript.* In Proceedings of the 9th Scheme and Functional Programming Workshop, 2008. (Cited on page 20.)

[Cohen 1977] Ellis Cohen. *Information Transmission in Computational Systems.* In Proceedings of the 6th ACM Symposium on Operating Systems Principles, SOSP '77, pages 133–139. ACM Press, 1977. (Cited on page 28.)

[Cousot 1977] Patrick Cousot and Radhia Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* In Proceedings of the Fourth ACM Symposium on Principles of Programming Languages (POPL'77), pages 238–252. ACM Press, 1977. (Cited on page 3.)

[Crockford] Douglas Crockford. *ADSafe*. http://www.adsafe.org. (Not cited.)

[Crockford 2008] Douglas Crockford. Javascript: The good parts. O'Reilly, 2008. (Cited on pages 20 and 51.)

[Davey 2002] Brian A. Davey and Hilary A. Priestley. Introduction to lattices and order (2. ed.). Cambridge University Press, 2002. (Cited on pages 3 and 81.)

[Denning 1976] Dorothy E. Denning. *A Lattice Model of Secure Information Flow*. Commun. ACM, vol. 19, no. 5, pages 236–243, 1976. (Cited on page 28.)

[Devriese 2010] Dominique Devriese and Frank Piessens. *Noninterference through Secure Multi-execution*. In Proceedings of the 31st IEEE Symposium on Security and Privacy, SP'10, pages 109–124. IEEE Computer Society, 2010. (Cited on pages 31 and 48.)

[Disney 2011] Tim Disney and Cormac Flanagan. *Gradual Information Flow Typing*. In STOP'11, 2011. (Cited on page 73.)

[Djoko 2008] Simplice Djoko, Rémi Douence and Pascal Fradet. *Specialized Aspect Languages Preserving Classes of Properties*. In Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods, pages 227–236. IEEE Computer Society, 2008. (Cited on pages 84 and 85.)

[FBJS] The FaceBook Team: FBJS. http://wiki.developers.facebook.com/index.php/FBJS. (Not cited.)

[Feldthaus 2014] Asger Feldthaus and Anders Møller. *Checking Correctness of TypeScript Interfaces for JavaScript Libraries*. In Proceedings of the 29th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, 2014. (Cited on page 73.)

[Fennell 2013] Luminous Fennell and Peter Thiemann. *Gradual Security Typing with References*. In Proceedings of the 26th IEEE Computer Security Foundations Symposium, CSF'26, pages 224–239. IEEE Computer Society, 2013. (Cited on page 73.)

[Flanagan 2011] David Flanagan. Javascript - the definitive guide. O'Reilly, 2011. (Cited on page 10.)

[Fragoso Santos 2014] José Fragoso Santos. *Online Materials - Inlining Compiler + Hybrid Type System*. http://www-sop.inria.fr/members/Jose.Santos/, 2014. (Cited on pages 32 and 49.)

[Gardner 2008] Philippa Gardner, Gareth Smith, Mark J. Wheelhouse and Uri Zarfaty. *DOM: Towards a Formal Specification*. In Proceedings of the ACM SIGPLAN Workshop PLAN-X on Programming Language Technologies for XML. ACM Press, 2008. (Cited on page 113.)

[Gardner 2012] Philippa Gardner, Sergio Maffeis and Gareth Smith. *Towards a program logic for JavaScript*. In Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, pages 31–44. ACM Press, 2012. (Cited on page 118.)

[Goguen 1982] Joseph A. Goguen and José Meseguer. *Security Policies and Security Models*. In Proceedings of the 3rd IEEE Symposium on Security and Privacy, SP'82, pages 11–20. IEEE Computer Society, 1982. (Cited on pages 3, 28 and 73.)

[Grosskurth 2005] Alan Grosskurth and Michael W. Godfrey. *A Reference Architecture for Web Browsers.* In Proceedings of the 21st International Conference on Software Maintenance, ICSM '05, pages 661–664. IEEE Computer Society, 2005. (Cited on pages 4, 89 and 113.)

[Guernic 2007] Gurvan Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses.* PhD thesis, Kansas State University, 2007. (Cited on pages 47 and 72.)

[Guha 2010] Arjun Guha, Claudiu Saftoiu and Shriram Krishnamurthi. *The Essence of Javascript.* In Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science, pages 126–150. Springer, 2010. (Cited on pages 20 and 21.)

[Guha 2012] Arjun Guha, Benjamin Lerner, Joe Gibbs Politz and Shriram Krishnamurthi. *Web API Verification: Results and Challenges.* 2012. (Cited on pages 4 and 75.)

[Hedin 2011] Daniel Hedin and Andrei Sabelfeld. *A Perspective on Information Flow Control.* Marktoberdorf, 2011. (Cited on pages 3 and 117.)

[Hedin 2012] Daniel Hedin and Andrei Sabelfeld. *Information-Flow Security for a Core of JavaScript.* In Proceedings of the 25th IEEE Computer Security Foundations Symposium, CSF'12, pages 3–18. IEEE Computer Society, 2012. (Cited on pages 25, 28, 29, 31, 33, 48 and 113.)

[Hedin 2014] Daniel Hedin, Arnar Birgisson, Luciano Bello and Andrei Sabelfeld. *JSFlow: Tracking Information Flow in JavaScript and its APIs.* In Proceedings of the 29th Symposium on Applied Computing, pages 1663–1671. ACM Press, 2014. (Cited on pages 4, 84, 113 and 117.)

[Hunt 2006] Sebastian Hunt and David Sands. *On Flow-sensitive Security Types.* In Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '06, pages 79–90. ACM Press, 2006. (Cited on pages 3, 32 and 119.)

[Jang 2009] Dongseok Jang and Kwang-Moo Choe. *Points-to Analysis for JavaScript.* In Proceedings of the 24th ACM Symposium on Applied Computing, pages 1930–1937. ACM Press, 2009. (Cited on page 20.)

[Jang 2010] Dongseok Jang, Ranjit Jhala, Sorin Lerner and Hovav Shacham. *An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications.* In Proceedings of the 17th ACM Conference on Computer and Communications Security, pages 270–283. ACM Press, 2010. (Cited on page 1.)

[Jensen 2009] Simon Holm Jensen, Anders Møller and Peter Thiemann. *Type Analysis for JavaScript.* In Proceedings of the 16th International Static Analysis Symposium (SAS), volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009. (Cited on pages 20, 73 and 74.)

[Keil 2013] Matthias Keil and Peter Thiemann. *Type-based dependency analysis for JavaScript.* In Proceedings of the 8th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, pages 47–58. ACM Press, 2013. (Cited on page 74.)

[Kiczales 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin. *Aspect-Oriented Programming.* In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP), pages 220–242, 1997. (Cited on page 84.)

[Li 2003] Peng Li, Yun Mao and S. Zdancewic. *Information Integrity Policies.* In Proceedings Formal Aspects in Security & Trust (FAST), 2003. (Cited on page 3.)

[Louw 2012] Mike Ter Louw, Karthik Thotta Ganesh and V. N. Venkatakrishnan. *AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements.* In Proceedings of the 19th USENIX Security Symposium, pages 371–388. USENIX Association, 2012. (Cited on page 2.)

[Luo 2012] Zhengqin Luo and Tamara Rezk. *Mashic Compiler: Mashup Sandboxing based on Inter-frame Communication.* In 25th IEEE Computer Security Foundations Symposium, pages 157–170. IEEE Computer Society, 2012. (Cited on pages 2 and 20.)

[Maffeis 2008] Sergio Maffeis, John C. Mitchell and Ankur Taly. *An Operational Semantics for JavaScript.* In Proceedings of the 6th Asian Symposium on Programming Languages and Systems, volume 5356 of *Lecture Notes in Computer Science*, pages 307–325. Springer, 2008. (Cited on pages 15, 20 and 21.)

[Maffeis 2009] Sergio Maffeis and Ankur Taly. *Language-Based Isolation of Untrusted JavaScript.* In Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF'09, pages 77–91. IEEE Computer Society, 2009. (Cited on pages 4, 53, 54, 73, 74 and 117.)

[Magazinius 2010a] Jonas Magazinius, Aslan Askarov and Andrei Sabelfeld. *A Lattice-based Approach to Mashup Security.* In Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS '10), pages 15–23. ACM Press, 2010. (Cited on page 2.)

[Magazinius 2010b] Jonas Magazinius, Phu H. Phung and David Sands. *Safe Wrappers and Sane Policies for Self Protecting JavaScript.* In Nordic Conference in Secure IT Systems, Lecture Notes in Computer Science, pages 239–255. Springer, 2010. (Cited on page 49.)

[Magazinius 2010c] Jonas Magazinius, Alejandro Russo and Andrei Sabelfeld. *On-the-fly Inlining of Dynamic Security Monitors.* In Proceedings of the 25th IFIP TC-11 International Information Security Conference, volume 330 of *IFIP Advances in Information and Communication Technology*, pages 173–186. Springer, 2010. (Cited on page 49.)

[Magazinius 2012] Jonas Magazinius, Alejandro Russo and Andrei Sabelfeld. *On-the-fly Inlining of Dynamic Security Monitors.* Computers & Security, vol. 31, no. 7, pages 827–843, 2012. (Cited on pages 31, 43 and 49.)

[Matthews 2009] Jacob Matthews and Robert Bruce Findler. *Operational Semantics for Multi-language Programs.* ACM Trans. Program. Lang. Syst., vol. 31, no. 3, pages 12:1–12:44, 2009. (Cited on page 84.)

[Microsoft 2014] Microsoft. *TypeScript language specification.* Rapport technique, Microsoft, 2014. (Cited on page 73.)

[Moore 2011] Scott Moore and Stephen Chong. *Static Analysis for Efficient Hybrid Information-Flow Control.* In Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF'24, pages 146–160. IEEE Computer Society, 2011. (Cited on page 72.)

[Phung 2009] Phu H. Phung, David Sands and Andrey Chudnov. *Lightweight Self-Protecting JavaScript.* In Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security (ASIACCS'09), pages 47–60. ACM Press, 2009. (Cited on page 49.)

[Politz 2011] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha and Shriram Krishnamurthi. *ADsafety: Type-Based Verification of JavaScript Sandboxing.* In Proceedings of the 20th USENIX Security Symposium. USENIX Association, 2011. (Cited on pages 73 and 74.)

[Pottier 2002] François Pottier and Vincent Simonet. *Information flow inference for ML.* In Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 319–330. ACM Press, 2002. (Cited on pages 3, 59 and 72.)

[Ramsey 2011] Norman Ramsey. *Embedding an interpreted language using higher-order functions and types.* J. Funct. Program., vol. 21, no. 6, pages 585–615, 2011. (Cited on page 84.)

[Richards 2010] Gregor Richards, Sylvain Lebresne, Brian Burg and Jan Vitek. *An Analysis of the Dynamic Behaviour of JavaScript Programs.* In Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10), volume 45, pages 1–12. ACM Press, 2010. (Not cited.)

[Russo 2008] Alejandro Russo, Koen Claessen and John Hughes. *A library for light-weight information-flow security in haskell.* In Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, pages 13–24. ACM Press, 2008. (Cited on page 48.)

[Russo 2009] Alejandro Russo, Andrei Sabelfeld and Andrey Chudnov. *Tracking Information Flow in Dynamic Tree Structures.* In Proceedings 14th European Symposium on Research in Computer Security, volume 5789 of *Lecture Notes in Computer Science*, pages 86–103. Springer, 2009. (Cited on pages vi, 5, 6, 89, 90, 112, 113, 114 and 115.)

[Russo 2010] Alejandro Russo and Andrei Sabelfeld. *Dynamic vs. Static Flow-Sensitive Security Analysis.* In Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF'10, pages 186–199. IEEE Computer Society, 2010. (Cited on pages 47, 48, 49 and 72.)

[Sabelfeld 2001] Andrei Sabelfeld and David Sands. *A Per Model of Secure Information Flow in Sequential Programs.* Higher Order and Symbolic Computation, vol. 14, no. 1, pages 59–91, 2001. (Cited on page 118.)

[Sabelfeld 2003a] Andrei Sabelfeld and Andrew C. Myers. *Language-Based Information-Flow Security.* IEEE Journal on Selected Areas in Communications, vol. 21, no. 1, pages 5–19, 2003. (Cited on pages 3, 4, 32, 48, 56 and 117.)

[Sabelfeld 2003b] Andrei Sabelfeld and Andrew C. Myers. *A Model for Delimited Information Release.* In Proceedings of the 9th Asian Symposium on Programming Languages and Systems, Lecture Notes in Computer Science, pages 220–237. Springer, 2003. (Cited on page 2.)

[Shroff 2007] Paritosh Shroff, Scott F. Smith and Mark Thober. *Dynamic Dependency Monitoring to Secure Information Flow.* In Proceedings of the 20th IEEE Computer Security Foundations Symposium, CSF'07, pages 203–217. IEEE Computer Society, 2007. (Cited on pages 47 and 72.)

[Smith 2011] Gareth Smith. *Local Reasoning about Web Programs.* PhD thesis, Imperial College London, 2011. (Cited on page 113.)

[Stefan 2011] Deian Stefan, Alejandro Russo, John C. Mitchell and David Mazières. *Flexible dynamic information flow control in Haskell.* In Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, pages 95–106, 2011. (Cited on page 48.)

[Stefan 2014] Deian Stefan, Amit Levy, Alejandro Russo and David Mazières. *Building secure systems with LIO (demo).* In Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell, pages 93–94, 2014. (Cited on page 48.)

[Taly 2011] Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller and Jasvir Nagra. *Automated Analysis of Security-Critical JavaScript APIs.* In Proceedings of the 32nd IEEE Symposium on Security and Privacy, pages 363–378. IEEE Computer Society, 2011. (Cited on pages 54, 83 and 84.)

[Thiemann 2005] Peter Thiemann. *Towards a Type System for Analysing JavaScript Programs.* In Proceedings of the 14th European Symposium on Programming Languages and Systems, Lecture Notes in Computer Science, pages 408–422. Springer, 2005. (Cited on pages 20, 56 and 73.)

[Venkatakrishnan 2006] Venkat N. Venkatakrishnan, Wei Xu, Daniel C. DuVarney and R. Sekar. *Provably Correct Runtime Enforcement of Non-interference Properties.* In Proceedings of 8th International Conference on Information and Communications Security, Lecture Notes in Computer Science, pages 332–351. Springer, 2006. (Cited on pages 47 and 72.)

[Volpano 1996] Dennis M. Volpano, Cynthia E. Irvine and Geoffrey Smith. *A Sound Type System for Secure Flow Analysis.* Journal of Computer Security, vol. 4, no. 2-3, pages 167–187, 1996. (Cited on pages 3, 28, 42, 72 and 117.)

[W3C Recommendation 2000] W3C Recommendation. *DOM: Document Object Model (DOM) Level 1 Specification (2nd Ed.).* Rapport technique, W3C, 2000. (Cited on page 4.)

[W3C Recommendation 2005] W3C Recommendation. *DOM: Document Object Model (DOM).* Rapport technique, W3C, 2005. (Cited on pages 4, 89, 91, 103, 113 and 114.)

[Yang 2013] Edward Yang, Deian Stefan, John Mitchell, David Mazières, Petr Marchenko and Brad Karp. *Toward Principled Browser Security.* In 14th Workshop on Hot Topics in Operating Systems. USENIX Association, 2013. (Cited on pages 2 and 3.)

[Zdancewic 2002] Stephan Zdancewic. *Programming Languages for Information Security.* PhD thesis, Cornell University, Ithaca, New York, 2002. (Cited on pages 32, 37, 47 and 97.)

[Zdancewic 2004] Steve Zdancewic. *Challenges for information-flow security.* In Proceedings of the 1st International Workshop on Programming Language Interference and Dependence, 2004. (Cited on page 117.)

Appendix A

Proofs of Chapter 4

A.1 Noninterference - Security Montior

The security monitor presented in Chapter was designed in such a way that the computed reading effect of an expression is always higher than or equal to the level of the program counter. Lemma A.1 formally states this property of the monitor.

Lemma A.1 (PC-Conservation)**.** *Given an expression e, a memory μ, a labelling Σ, a level σ_{pc}, and a reference r such that: $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle$ for some memory μ', value v, labelling Σ', and security level σ; then it is always the case that $\sigma_{pc} \sqsubseteq \sigma$.*

Proof: The result follows by induction on the derivation of $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle$. It suffices to note that the rules [Value], [This], [Variable], [Property Deletion], and [Object Literal] explicitly set the reading effect of the current expression to a level higher than or equal to the level of the program counter. All the other rules set the reading effect of the current expression to a level higher than or equal to the reading effect of one of its subexpressions. Applying the induction hypothesis, the result immediately follows. □

A.1.1 Proving Confinement

In order to prove that security monitor is confined, one first needs to state for each type of operation that modifies the memory, the conditions under which that type of operation is *confined*. In other words, the conditions under which that type of operation does not modify *low memory*. We identify four types of operations that change the memory:

- **Property Assignment.** A property assignment changes the memory either by creating a new property in an existing object or by updating the value of an existing property of an existing object. Proposition A.1 states that a **property update** is confined if the *value level* of the updated property is not observable, whereas a **property creation** is confined if the *existence level* of the created property is not observable.
- **Property Deletion.** A property deletion changes the memory by deleting an existing property in an existing object. Proposition A.2 states that a property deletion is confined if the *existence level* of the deleted property is not observable.
- **Object Creation.** Proposition A.3 states that an object creation is confined if both the the object level and the structure security level of the created object are not observable.
- **Scope Allocation.** Proposition A.4 states that the allocation of a scope object is not observable provided that the level of the context in which the body of the function to be executed is not observable.

Proposition A.1 (Confined Property Assignment)**.** *Given two memories μ and μ', respectively labelled by Σ and Σ', a reference r, a property p, a value v, and three security levels $\sigma, \sigma', \sigma'' \in \mathcal{L}$, such that:* **(1)** *$\mu' = \mu[r \cdot p \mapsto v]$,* **(2)** *$\Sigma' = \mathsf{updt}(\Sigma, (r, p), (\sigma', \sigma''))$, and* **(3)** *$p \notin dom(\mu(r)) \Rightarrow \sigma' \sqcap \sigma'' \sqcap \Sigma.\mathsf{struct}(r) \not\sqsubseteq \sigma$, and* **(4)** *$p \in dom(\mu(r)) \Rightarrow \sigma'' \sqcap \Sigma.\mathsf{val}(r \cdot p) \not\sqsubseteq \sigma$; then, it follows that $\mu, \Sigma \sim_\sigma \mu', \Sigma'$.*

Proof: The result follows immediately from the definitions of low-equality and updt. □

Proposition A.2 (Confined Property Deletion)**.** *Given two memories μ and μ', respectively labelled by Σ and Σ', a reference r, a property p, and a security level $\sigma \in \mathcal{L}$, such that:* **(1)** $\mu' = \mu\left[r \mapsto \mu(r)|_{dom(\mu(r))\setminus p}\right]$, **(2)** $\Sigma' = \mathsf{contract}(\Sigma, r, p)$, *and* **(3)** $\Sigma.\mathsf{exist}(r \cdot p) \not\sqsubseteq \sigma$*; then, it follows that $\mu, \Sigma \sim_\sigma \mu', \Sigma'$.*

Proof: The result follows immediately from the definitions of low-equality and contract. □

Proposition A.3 (Confined Object Creation)**.** *Given two memories μ and μ', respectively labelled by Σ and Σ', a reference $r \notin dom(\mu)$, and three security levels $\sigma, \sigma_o, \sigma_s \in \mathcal{L}$, such that:* **(1)** $\mu' = \mu\left[r \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}]\right]$, **(2)** $\Sigma' = \mathsf{updt}(\Sigma'', (r, \texttt{"\_prot\_"}), (\sigma_o, \sigma_o))$ *where* $\Sigma'' = \mathsf{extend}(\Sigma, r, \sigma_o, \sigma_s)$, *and* **(3)** $\sigma_o \sqcap \sigma_s \not\sqsubseteq \sigma$*; then, it follows that $\mu, \Sigma \sim_\sigma \mu', \Sigma'$.*

Proof: The result follows immediately from the definitions of low-equality. □

Proposition A.4 (Confined Scope Allocation)**.** *Given two memories μ and μ', respectively labelled by Σ and Σ', three references $r_f, r_{this}, r_{scope} \in \texttt{Ref}$, a value v_{arg}, and three security levels $\sigma, \sigma_{arg}, \sigma_{pc} \in \mathcal{L}$, such that:* **(1)** $\langle \mu', e, r_{scope}, \Sigma' \rangle = \mathsf{NewScope}(\mu, r_f, v_{arg}, r_{this}, \Sigma, \sigma_{pc}, \sigma_{arg})$ *and* **(2)** $\sigma_{pc} \sqcap \sigma_{arg} \not\sqsubseteq \sigma$*; then, it follows that $\mu, \Sigma \sim_\sigma \mu', \Sigma'$.*

Proof: The result follows immediately from the definitions of low-equality and $\mathsf{NewScope}_{lab}$. □

Finally, below, we present the proof of the main confinement theorem.

Lemma 4.1 - Confinement

Proof: Hypothesis of the Lemma:

- $\sigma_{pc}, r \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle$ (hyp.1)
- $\sigma_{pc} \not\sqsubseteq \sigma'$ (hyp.2)

The claim of the lemma is that: $\mu, \Sigma \sim_{\sigma'} \mu', \Sigma'$. The proof proceeds by induction on the derivation of (hyp.1). We distinguish two types of base cases:

- Those that do neither change the memory nor the labeling: [Value], [This], and [Variable]. Since in all of these cases $\mu' = \mu$ and $\Sigma = \Sigma'$, it immediately follows that: $\mu, \Sigma \sim_{\sigma'} \mu', \Sigma'$.
- Those that change the heap by adding a new object: [Function Literal] and [Object Literal].

Analogously, we distinguish three types of inductive cases:

1. Those that do not directly change the memory: [Binary Operation], [Property Look-up], [Membership Testing], [Sequence], and [Conditional].
2. Those that directly change the memory by allocating a new object: [Function Call] and [Method Call].
3. Those that directly change the memory either by creating a new property, updating the value of an existing property, or by deleting an existing property: [Variable Assignment], [Property Assignment], and [Property Deletion].

We prove one case of each type (the others are analogous).

[Function Literal] Suppose that $e = \mathsf{function}(x)\{\mathsf{var}\ y_1, \cdots, y_n;\ e\}$ (hyp.3). We conclude that there is a reference r_f and two labellings Σ_0 and Σ_1 such that:

- $\mu' = \mu\,[r' \mapsto [\texttt{"@fscope"} \mapsto r, \texttt{"@code"} \mapsto \lambda x.\,\{\mathsf{var}\ y_1, \cdots, y_n;\ e\}]]$ (1) - (hyp.1) + (hyp.3)
- $\Sigma_0 = \mathsf{extend}(\Sigma, r_f, \sigma_{pc}, \sigma_{pc})$, $\Sigma_1 = \mathsf{updt}(\Sigma_0, (r_f, \texttt{"@fscope"}), (\sigma_{pc}, \sigma_{pc}))$, and $\Sigma' = \mathsf{updt}(\Sigma_1, (r_f, \texttt{"@code"}), (\sigma_{pc}, \sigma_{pc}))$, (2) - (hyp.1) + hyp3
- $\mu, \Sigma \sim_{\sigma'} \mu', \Sigma'$ **(3)** - (hyp.2) + (1) + (2)

[Property Assignment] Suppose that $e = e_0[e_1] = e_2$ (hyp.3). We conclude that there are three memories μ_0, μ_1, and μ_2, three labelings Σ_0, Σ_1, and Σ_2, a reference r_0, a string $m_1 \in \texttt{Str}$, and three security levels σ_0, σ_1, and σ_2 such that:

- $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma\rangle \Downarrow_{IF} \langle \mu_0, r_0, \Sigma_0, \sigma_0\rangle$ (1) - (hyp.1) + (hyp.3)
- $r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0\rangle \Downarrow_{IF} \langle \mu_1, m_1, \Sigma_1, \sigma_1\rangle$ (2) - (hyp.1) + (hyp.3)
- $r, \sigma_{pc} \vdash \langle \mu_1, e_2, \Sigma_1\rangle \Downarrow_{IF} \langle \mu_2, v_2, \Sigma_2, \sigma_2\rangle$ (3) - (hyp.1) + (hyp.3)
- $\mu, \Sigma \sim_{\sigma'} \mu_0, \Sigma_0$ (4) - (hyp.2) + (1) + **ih**
- $\mu_0, \Sigma_0 \sim_{\sigma'} \mu_1, \Sigma_1$ (5) - (hyp.2) + (2) + **ih**
- $\mu_1, \Sigma_1 \sim_{\sigma'} \mu_2, \Sigma_2$ (6) - (hyp.2) + (3) + **ih**
- $\mu, \Sigma \sim_{\sigma'} \mu_2, \Sigma_2$ (7) - (4) - (6) + Transitivitiy of $\sim_{\sigma'}$
- $\sigma_{pc} \sqsubseteq \sigma_0 \sqcap \sigma_1 \sqcap \sigma_2$ (8) - (1) - (3) + PC-Conservation (Lemma A.1)
- $\sigma_0 \sqcap \sigma_1 \sqcap \sigma_2 \not\sqsubseteq \sigma'$ (9) - (hyp.2) + (8)
- $\mu' = \mu_2[r_0 \cdot m_1 \mapsto v_2]$ (10) - (hyp.1) + (hyp.3)
- $\Sigma' = \mathsf{updt}(\Sigma_3, (v_0, v_1), (\sigma_0 \sqcup \sigma_1, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2))$ (11) - (hyp.1) + (hyp.3)
- Case $m_1 \in \mu_2(r_0)$ ((hyp.4)):
  - $\sigma_0 \sqcup \sigma_1 \sqsubseteq \Sigma_2.\mathsf{val}(r_0 \cdot m_1)$ (12.1) - (hyp.1) + (hyp.3) + (hyp.4)
  - $\Sigma_2.\mathsf{val}(r_0 \cdot m_1) \not\sqsubseteq \sigma'$ (12.2) - (9) + (12.1)
  - $\mu_2, \Sigma_2 \sim_{\sigma'} \mu', \Sigma'$ (12.3) - (10) - (11) + (12.2) + Confined Property Assignment (Proposition A.1)
  - $\mu, \Gamma, \Sigma \sim_{\sigma'} \mu', \Sigma'$ (12.4) - (7) + (12.3) + Transitivity of $\sim_{\sigma'}$
- Case $m_1 \notin \mu_2(r_0)$ ((hyp.4)):
  - $\sigma_0 \sqcup \sigma_1 \sqsubseteq \Sigma_2.\mathsf{struct}(r_0)$ (13.1) - (hyp.1) + (hyp.3) + (hyp.4)
  - $\Sigma_2.\mathsf{struct}(r_0) \not\sqsubseteq \sigma'$ (13.2) - (9) + (13.1)
  - $\mu_2, \Sigma_2 \sim_{\sigma'} \mu', \Sigma'$ (13.3) - (10) - (11) + (13.2) + Confined Property Assignment (Proposition A.1)
  - $\mu, \Sigma \sim_{\sigma'} \mu', \Sigma'$ (13.4) - (7) + (13.3) + Transitivity of $\sim_{\sigma'}$
- $\mu, \Sigma \sim_{\sigma'} \mu', \Sigma'$ **(14)** - (12) + (13)

[Function Call] Suppose that $e = e_0(e_1)^i$ (hyp.3). We conclude that there are three memories μ_0, μ_1, and $\hat{\mu}$, three labellings Σ_0, Σ_1, and $\hat{\Sigma}$, a reference r_0, a value v_1, four security levels σ_0, σ_1, σ_2, and $\hat{\sigma}$, and an expression $\hat{e}$ such that:

- $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma\rangle \Downarrow_{IF} \langle \mu_0, r_0, \Sigma_0, \sigma_0\rangle$ (1) - (hyp.1) + (hyp.3)
- $r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0\rangle \Downarrow_{IF} \langle \mu_1, v_1, \Sigma_1, \sigma_1\rangle$ (2) - (hyp.1) + (hyp.3)
- $\langle \hat{r}, \hat{\mu}, \hat{e}, \hat{\Sigma}\rangle = \mathsf{NewScope}(\mu_1, r_0, v_1, \#glob, \Sigma_1, \sigma_0, \sigma_1)$ (3) - (hyp.1) + (hyp.3)

- $\hat{r}, \sigma_0 \vdash \langle \hat{\mu}, \hat{e}, \hat{\Sigma} \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma \rangle$ — (4) - (hyp.1) + (hyp.3)
- $\mu, \Sigma \sim_{\sigma'} \mu_0, \Sigma_0$ — (5) - (hyp.2) + (1) + **ih**
- $\mu_0, \Sigma_0 \sim_{\sigma'} \mu_1, \Sigma_1$ — (6) - (hyp.2) + (2) + **ih**
- $\mu, \Sigma \sim_{\sigma'} \mu_1, \Sigma_1$ — (7) - (5) + (6) + Transitivitiy of $\sim_{\sigma'}$
- $\sigma_{pc} \sqsubseteq \sigma_0 \sqcap \sigma_1$ — (8) - (1) + (2) + PC-Level-Conservation Lemma
- $\sigma_0 \sqcap \sigma_1 \not\sqsubseteq \sigma'$ — (9) - (hyp.2) + (8)
- $\mu_1, \Sigma_1 \sim_{\sigma'} \hat{\mu}, \hat{\Sigma}$ — (10) - (3) + (9) + Confined Scope Allocation (Proposition A.4)
- $\hat{\mu}, \hat{\Sigma} \sim_{\sigma'} \mu', \Sigma'$ — (11) - (4) + (9) + **ih**
- $\mu, \Sigma \sim_{\sigma'} \mu', \Sigma'$ — **(13)** - (7) + (10) + (11) + Transitivity of $\sim_{\sigma'}$

□

A.1.2 Proving Noninterference

In order to prove noninterference, it is useful to establish some intermediate results to reason about the conditions under which observable operations that change the memory preserve the low-equality relation. To this end, we start by establishing two indistinguishability results concerning the scope-chain and the prototype-chain look-up procedures. Concretely, Lemma A.2 states that the results of applying the scope-chain look-up procedure in two low-equal memories in visible scopes are the same. Lemma A.3 states that the results of applying the prototype-chain look-up procedure in two low-equal memories are low-equal. That is, either both results are observable and coincide or they are both unobservable.

Lemma A.2 (Scope-Chain Indistinguishability)**.** *Given two memories μ_0 and μ_1 respectively labelled by Σ_0 and Σ_1, a reference r, a security level σ, and a string $m \in \texttt{Str}$ such that:* **(1)** $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$, **(2)** $r_0 = \mathsf{Scope}(\mu_0, r, m)$, **(3)** $r_1 = \mathsf{Scope}(\mu_1, r, m)$, *and* **(4)** $\Sigma_0.\mathsf{obj}(r) \sqcup \Sigma_1.\mathsf{obj}(r) \sqsubseteq \sigma$*; it follows that:* $r_0 = r_1$.

Proof: We restate the hypotheses: $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$ (hyp.1), $r_0 = \mathsf{Scope}(\mu_0, r, m)$ (hyp.2), $r_1 = \mathsf{Scope}(\mu_1, r, m)$ (hyp.3), $\Sigma_0.\mathsf{struct}(r) \sqcup \Sigma_1..\mathsf{struct}(r) \sqsubseteq \sigma$ (hyp.4). We proceed by induction on the derivation of $r_0 = \mathsf{Scope}(\mu_0, r, x)$. The base cases are [Null] and [Base], whereas the inductive case is [Look-up].

[Null] Suppose that $r = \texttt{null}$ (hyp.5). We conclude that:

- $r_0 = r_1 = \mathsf{null}$ — **(1)** - (hyp.2) + (hyp.3) + (hyp.6)

[Base] Suppose that $m \in dom(\mu_0(r_0))$ (hyp.5). We conclude that:

- $r_0 = r$ — (1) - (hyp.3) + (hyp.5)
- $dom(\mu_0(r)) = dom(\mu_1(r))$ — (2) - (hyp.1) + (hyp.4)
- $m \in dom(\mu_1(r))$ — (3) - (hyp.5) + (2)
- $r_1 = r$ — (4) - (hyp.3) + (3)
- $r_0 = r$ — **(5)** - (hyp.2) + (1) + (4)

[Look-up] Suppose that $m \notin dom(\mu_0(r))$ (hyp.5) and $r \neq \texttt{null}$ (hyp.6). We conclude that:

- $r_0 = \mathsf{Scope}(\mu_0, r'_0, m)$, where: $r'_0 = \mu_0(r \cdot \texttt{"@scope"})$ — (1) - (hyp.2) + (hyp.5) + (hyp.6)
- $dom(\mu_0(r)) = dom(\mu_1(r))$ — (2) - (hyp.1) + (hyp.4)

- $m \notin dom(\mu_1(r))$ (3) - (hyp.5) + (2)
- $r_1 = \mathsf{Scope}(\mu_1, r'_1, m)$, where: $r'_1 = \mu_1(r_1 \cdot \texttt{"@scope"})$ (4) - (hyp.4) + (3)
- $\Sigma_i.\mathsf{struct}(r'_i) \sqsubseteq \Sigma_i.\mathsf{struct}(r) = \Sigma_i.\mathsf{val}(r_i \cdot \texttt{"@scope"})$ for $i = 0, 1$ (5) - (1) + (4) + Well-Labelled Scope-Chains
- $\Sigma_i.\mathsf{val}(r_i \cdot \texttt{"@scope"}) \sqsubseteq \sigma$, for $i = 0, 1$ (6) - (hyp.4) + (5)
- $r'_0 = r'_1$ (7) - (hyp.1) + (6)
- $\Sigma_i.\mathsf{struct}(r'_i) \sqsubseteq \sigma$, for $i = 0, 1$ (8) - (hyp.4) + (5)
- $r_0 = r'_1$ **(9)** - (hyp.1) + (1) + (4) + (7) + (8) + **ih**

□

Lemma A.3 (Prototype-Chain Indistinguishability)**.** *Given two memories μ_0 and μ_1 respectively labelled by Σ_0 and Σ_1, a reference r, a security level σ, and a string $m \in \texttt{Str}$ such that:* **(1)** $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$, **(2)** $\langle r_0, \sigma_0 \rangle = \mathsf{Proto}(\mu_0, r, m, \Sigma_0)$, *and* **(3)** $\langle r_1, \sigma_1 \rangle = \mathsf{Proto}(\mu_1, r, m, \Sigma_1)$*; it holds that:* $r_0, \sigma_0 \sim_\sigma r_1, \sigma_1$.

Proof: We restate the hypotheses: $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$ (hyp.1), $\langle r_0, \sigma_0 \rangle = \mathsf{Proto}(\mu_0, r, m, \Sigma_0)$ (hyp.2), and $\langle r_1, \sigma_1 \rangle = \mathsf{Proto}(\mu_1, r, m, \Sigma_1)$ (hyp.3). To prove the result one has to prove that the implication:

$$\sigma_i \sqsubseteq \sigma \Rightarrow (r_0 = r_1 \wedge \sigma_0 = \sigma_1)$$

holds for $i = 0, 1$. We prove the result for $i = 0$. The proof for $i = 1$ is symmetric. We proceed by induction on the derivation of (hyp.2) and we assume that $\sigma_0 \sqsubseteq \sigma$ (hyp.4). The base cases are [Null] and [Base], whereas the inductive case is [Look-up].

[Null] Suppose that $r = \mathsf{null}$ (hyp.5). We conclude that:

- $r_0 = \mathsf{null}$ and $\sigma_0 = \bot$ (1) - (hyp.2) + (hyp.5)
- $r_1 = \mathsf{null}$ and $\sigma_1 = \bot$ (2) - (hyp.3) + + (hyp.5)
- $r_0 = r_1$ and $\sigma_0 = \sigma_1$ **(3)** - (1) + (2)

[Base] Suppose that $m \in dom(\mu_0(r))$ (hyp.5). We conclude that:

- $r_0 = r$ and $\sigma_0 = \Sigma_0.\mathsf{exist}(r \cdot m)$ (1) - (hyp.3) + (hyp.6)
- $\Sigma_0.\mathsf{exist}(r \cdot m) \sqsubseteq \sigma$ (2) - (hyp.4) + (1)
- $m \in dom(\mu_1(r))$ and $\Sigma_0(r \cdot m) = \Sigma_1(r \cdot m) \sqsubseteq \sigma$ (3) - (hyp.1) + (2)
- $r_1 = r$ and $\sigma_1 = \Sigma_1(r \cdot m) = \sigma_0$ (4) - (hyp.3) + (3)
- $r_0 = r_1$ and $\sigma_0 = \sigma_1 \sqsubseteq \sigma$ **(5)** - (1) + (4)

[Look-up] Suppose that $m \notin dom(\mu_0(r))$ (hyp.5) and $r \neq \mathsf{null}$ (hyp.6). We conclude that there is a security level σ'_0 such that:

- $\langle r_0, \sigma'_0 \rangle = \mathsf{Proto}(\mu_0, r'_0, m, \Sigma_0)$ and $\sigma_0 = \Sigma_0.\mathsf{val}(r \cdot \texttt{"\_prot\_"}) \sqcup \Sigma_0.\mathsf{struct}(r) \sqcup \sigma'_0$ where $r'_0 = \mu_0(r_0 \cdot \texttt{"\_prot\_"})$ (1) - (hyp.2) + (hyp.5) + (hyp.6)
- $\Sigma_0.\mathsf{struct}(r) \sqsubseteq \sigma$ (2) - (hyp.4) + (1)
- $dom(\mu_0(r)) = dom(\mu_1(r))$ and $\Sigma_0.\mathsf{struct}(r) = \Sigma_1.\mathsf{struct}(r) \sqsubseteq \sigma$ (3) - (hyp.1) + (2)
- $m \notin dom(\mu_1(r))$ (4) - (hyp.5) + (3)
- $\langle r_1, \sigma'_1 \rangle = \mathsf{Proto}(\mu_1, r'_1, m, \Sigma_1)$ and $\sigma_1 = \Sigma_1.\mathsf{val}(r \cdot \texttt{"\_prot\_"}) \sqcup \Sigma_1.\mathsf{struct}(r) \sqcup \sigma'_1$ where $r'_1 = \mu_1(r_1 \cdot \texttt{"\_prot\_"})$ (5) - (hyp.3) + (hyp.6) + (4)

- $\Sigma_0.\mathsf{val}(r \cdot \texttt{"\_prot\_"}) \sqsubseteq \sigma$ (6) - (hyp.4) + (1)
- $r'_0 = r'_1$ and $\Sigma_0.\mathsf{val}(r \cdot \texttt{"\_prot\_"}) = \Sigma_1.\mathsf{val}(r \cdot \texttt{"\_prot\_"}) \sqsubseteq \sigma$ (7) - (hyp.1) + (1) + (5) + (6)
- $\sigma'_0 \sqsubseteq \sigma \Rightarrow (r_0 = r_1 \wedge \sigma'_0 = \sigma'_1 \sqsubseteq \sigma)$ (8) - (hyp.1) + (1) + (5) + (7) + **ih**
- $\sigma'_0 \sqsubseteq \sigma$ (9) - (hyp.4) + (1)
- $\sigma'_0 = \sigma'_1 \sqsubseteq \sigma$ and $r_0 = r_1$ **(10)** - (8) + (9)
- $\sigma_1 = \sigma_0 \sqsubseteq \sigma$ **(11)** - (1) + (3) + (5) + (7) + (10)

□

In order to prove noninterference, it is useful to state for each type of operation that modifies the memory, the conditions under which, when performed in low-equal memories in observable contexts, they produce two low-equal memories. As we did for confinement, we consider each type of operation that modifies the memory individually.

- **Property Assignment.** Proposition A.5 states that if one assigns two low-equal values to the same property of two objects pointed to by the same reference in two low-equal memories, the resulting memories are still low-equal.
- **Property Deletion.** Proposition A.6 states that if one deletes the same property in two objects pointed to by the same reference in two low-equal memories, the resulting memories are still low-equal.
- **Object Creation.** Proposition A.7 states that the allocation of a new empty object in the same new reference in two low-equal memories yields two low-equal memories.
- **Scope Allocation.** Proposition A.8 states that the allocation of a new scope object in the same new reference in two-equal memories yields two low-equal memories.

Proposition A.5 (Noninterferent Property Assignment)**.** *Given four memories μ_0, μ'_0, μ_1, and μ'_1, respectively labelled by Σ_0, Σ'_0, Σ_1, and Σ'_1, a reference r, a property p, two values v_0 and v_1, and four security levels $\sigma, \sigma', \sigma_0, \sigma_1 \in \mathcal{L}$, such that:*

- $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$,
- $\mu'_0 = \mu_0[r \cdot p \mapsto v_0]$ *and* $\mu'_1 = \mu_1[r \cdot p \mapsto v_1]$,
- $\Sigma'_0 = \mathsf{updt}(\Sigma_0, (r, p), (\sigma', \sigma_0))$ *and* $\Sigma'_1 = \mathsf{updt}(\Sigma_1, (r, p), (\sigma', \sigma_1))$,
- $v_0, \sigma_0 \sim_\sigma v_1, \sigma_1$*;*

then, it follows that $\mu'_0, \Sigma'_0 \sim_\sigma \mu'_1, \Sigma'_1$.

Proof: The result follows immediately from the definitions of low-equality and updt. □

Proposition A.6 (Noninterferent Property Deletion)**.** *Given four memories μ_0, μ'_0, μ_1, and μ'_1, respectively labelled by Σ_0, Σ'_0, Σ_1, and Σ'_1, a reference r, a property p, and a security level σ, such that:*

- $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$,
- $\mu'_0 = \mu_0 \left[r \mapsto \mu_0(r)|_{dom(\mu_0(r)) \setminus p}\right]$ *and* $\mu'_1 = \mu_1 \left[r \mapsto \mu_1(r)|_{dom(\mu_1(r)) \setminus p}\right]$,
- $\Sigma'_0 = \mathsf{contract}(\Sigma_0, r, p)$ *and* $\Sigma'_1 = \mathsf{contract}(\Sigma_1, r, p)$*;*

then, it follows that $\mu_0', \Sigma_0' \sim_\sigma \mu_1', \Sigma_1'$.

Proof: The result follows immediately from the definitions of low-equality and contract. □

Proposition A.7 (Noninterferent Object Creation)**.** *Given four memories* μ_0, μ_0', μ_1, *and* μ_1', *respectively labelled by* Σ_0, Σ_0', Σ_1, *and* Σ_1', *and three security levels* $\sigma, \sigma_o, \sigma_s \in \mathcal{L}$, *such that:*

- $r \notin dom(\mu_0) \cup dom(\mu_1)$,
- $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$,
- $\mu_0' = \mu_0\,[r \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}]]$ *and* $\mu_1' = \mu_1\,[r \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}]]$,
- $\Sigma_0' = \mathsf{updt}(\Sigma_0'', (r, \texttt{"\_prot\_"}), (\sigma_o, \sigma_o))$ *and* $\Sigma_1' = \mathsf{updt}(\Sigma_1'', (r, \texttt{"\_prot\_"}), (\sigma_o, \sigma_o))$,

where $\Sigma_0'' = \mathsf{extend}(\Sigma_0, r, \sigma_o, \sigma_s)$ *and* $\Sigma_1'' = \mathsf{extend}(\Sigma_1, r, \sigma_o, \sigma_s)$*; then, it follows that* $\mu_0', \Sigma_0' \sim_\sigma \mu_1', \Sigma_1'$.

Proof: The result follows immediately from the definitions of low-equality. □

Proposition A.8 (Noninterferent Scope Allocation)**.** *Given four memories* μ_0, μ_0', μ_1, *and* μ_1', *respectively labelled by* Σ_0, Σ_0', Σ_1, *and* Σ_1', *three references* $r_f, r_{this}, r_{scope} \in \mathtt{Ref}$, *two values* v_{arg}^0 *and* v_{arg}^1, *and four security levels* $\sigma, \sigma_{arg}^0, \sigma_{arg}^1, \sigma_{pc} \in \mathcal{L}$, *such that:*

- $\sigma_{pc} \sqsubseteq \sigma$,
- $v_{arg}^0, \sigma_{arg}^0 \sim_\sigma v_{arg}^1, \sigma_{arg}^1$,
- $\langle \mu_0', e_0, r_{scope}^0, \Sigma_0' \rangle = \mathsf{NewScope}(\mu_0, r_f, v_{arg}^0, r_{this}, \Sigma_0, \sigma_{pc}, \sigma_{arg}^0)$,
- $\langle \mu_1', e_1, r_{scope}^1, \Sigma_1' \rangle = \mathsf{NewScope}(\mu_1, r_f, v_{arg}^1, r_{this}, \Sigma_1, \sigma_{pc}, \sigma_{arg}^1)$

then, it holds that $\mu_0', \Sigma_0' \sim_\sigma \mu_1', \Sigma_1'$, $r_{scope}^0 = r_{scope}^1$, *and* $e_0 = e_1$.

Proof: The result follows immediately from the definitions of low-equality and $\mathsf{NewScope}_{lab}$. □

The proof of the Noninterference Theorem requires the

Definition A.1 (Well-labelled (Function Objects))**.** *Given a memory* μ *labelled by* Σ, *the function objects in* μ *are said to be well-labelled by* Σ *if for every function object* o_f *in the range of* μ *pointed to by a reference* r_f, *it holds that:*

$$\forall_{r_s \in dom(\mu)}\ \mu(r_s \cdot m) = r_f \Rightarrow \begin{cases} \Sigma.\mathsf{exist}(r_f \cdot \texttt{"@code"}) \sqcup \Sigma.\mathsf{val}(r_f \cdot \texttt{"@code"}) \sqsubseteq \Sigma.\mathsf{val}(r_s \cdot m) \\ \Sigma.\mathsf{exist}(r_f \cdot \texttt{"@fscope"}) \sqcup \Sigma.\mathsf{val}(r_f \cdot \texttt{"@fscope"}) \sqsubseteq \Sigma.\mathsf{val}(r_s \cdot m) \end{cases}$$

Lemma A.4 (Well-labelled Memory)**.** *Given a memory* μ *labelled by* Σ, *a reference* r, *an expression* e, *and two security levels* σ_{pc} *and* σ, *such that:*

- *the function objects in* μ *are well-labelled by* Σ *(hyp.1),*
- $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$ *(hyp.2),*
- $\Sigma.\mathsf{struct}(r) \sqsubseteq \sigma_{pc}$ *(hyp.3)*

It holds that:

- *the function objects in* μ_f *are well-labelled by* Σ_f *and*
- $\Sigma_f.\mathsf{struct}(r) \sqsubseteq \sigma_{pc}$

Proof: The proof proceeds by induction on the derivation of (hyp.2). □

Finally, below, we present the proof of the main noninterference theorem.

Theorem 4.2 - Monitor Noninterference

Proof: We restate the hypotheses of the theorem:

- $\mu, \Sigma \sim_\sigma \mu', \Sigma'$ (hyp.1),
- $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$ (hyp.2),
- $r, \sigma_{pc} \vdash \langle \mu', e, \Sigma' \rangle \Downarrow_{IF} \langle \mu'_f, v'_f, \Sigma'_f, \sigma'_f \rangle$ (hyp.3).

If $\sigma_{pc} \not\sqsubseteq \sigma$, we apply the Confinement Lemma (Lemma 4.1) to (hyp.2) and (hyp.3) and conclude that $\mu, \Sigma \sim_\sigma \mu_f, \Sigma_f$ and $\mu', \Sigma' \sim_\sigma \mu'_f, \Sigma'_f$. Using the transitivity of $\sim_\sigma$, we conclude that $\mu', \Sigma' \sim_\sigma \mu'_f, \Sigma'_f$. Applying the PC-Conservation Lemma (Lemma A.1), we conclude that $\sigma_{pc} \sqsubseteq \sigma_f \sqcap \sigma'_f$. Since we are assuming that $\sigma_{pc} \not\sqsubseteq \sigma$, we conclude that both v_f and v'_f are not observable and the result follows.

In the following, we assume $\sigma_{pc} \sqsubseteq \sigma$ (hyp.4). We proceed by induction on the depth of the derivation tree of (hyp.2). With respect to the second claim of the theorem, in every case, we only prove $\sigma_f \sqsubseteq \sigma \Rightarrow v_f = v'_f \wedge \sigma_f = \sigma'_f \sqsubseteq \sigma$. The proof of the symmetric implication $\sigma'_f \sqsubseteq \sigma \Rightarrow v_f = v'_f \wedge \sigma_f = \sigma'_f \sqsubseteq \sigma$ is always done in the exact same way.

[Value] Suppose that $e = v$ (hyp.5). We conclude that:

- $r, \sigma_{pc} \vdash \langle \mu, v, \Sigma \rangle \Downarrow_{IF} \langle \mu, v, \Sigma, \sigma_{pc} \rangle$ (1) - (hyp.5)
- $r, \sigma_{pc} \vdash \langle \mu', v, \Sigma' \rangle \Downarrow_{IF} \langle \mu', v, \Sigma', \sigma_{pc} \rangle$ (2) - (hyp.5)
- $\mu_f = \mu$, $\Sigma_f = \Sigma$, $v_f = v$, and $\sigma_f = \sigma_{pc}$ (3) - (hyp.2) + (1)
- $\mu'_f = \mu'$, $\Sigma'_f = \Sigma'$, $v'_f = v'$, and $\sigma'_f = \sigma_{pc}$ (4) - (hyp.3) + (2)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(5)** - (hyp.1) + (3) + (4)
- $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$ **(6)** - (3) + (4)

[This] Suppose that $e = \mathsf{this}$ (hyp.5). We conclude that:

- $r, \sigma_{pc} \vdash \langle \mu, \mathsf{this}, \Sigma \rangle \Downarrow_{IF} \langle \mu, v_f, \Sigma, \sigma_f \rangle$, $v_f = \mu(r \cdot \texttt{"@this"})$, and $\sigma_f = \Sigma.\mathsf{val}(r \cdot \texttt{"@this"}) \sqcup \sigma_{pc}$ (1) - (hyp.2) + (hyp.5)
- $r', \sigma_{pc} \vdash \langle \mu', \mathsf{this}, \Sigma' \rangle \Downarrow_{IF} \langle \mu', v'_f, \Sigma', \sigma'_f \rangle$, $v'_f = \mu'(r' \cdot \texttt{"@this"})$, and $\sigma'_f = \Sigma'.\mathsf{val}(r \cdot \texttt{"@this"}) \sqcup \sigma_{pc}$ (2) - (hyp.3) + (hyp.5)
- $\mu_f = \mu$, $\Sigma_f = \Sigma$, $\mu'_f = \mu'$, and $\Sigma'_f = \Sigma'$ (3) - (hyp.2) + (hyp.3) + (1) + (2)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(4)** - (hyp.1) + (3)
- $\Sigma.\mathsf{val}(r \cdot \texttt{"@this"}) \sqsubseteq \sigma \Rightarrow (v_f = v'_f \ \wedge \ \Sigma.\mathsf{val}(r \cdot \texttt{"@this"}) = \Sigma'.\mathsf{val}(r \cdot \texttt{"@this"}) \sqsubseteq \sigma)$ (5) - (hyp.1) + (1) + (2)
- $\sigma_f \sqsubseteq \sigma \Rightarrow \Sigma.\mathsf{val}(r \cdot \texttt{"@this"}) \sqsubseteq \sigma$ (6) - (1)
- $\Sigma.\mathsf{val}(r \cdot \texttt{"@this"}) = \Sigma'.\mathsf{val}(r \cdot \texttt{"@this"}) \Rightarrow \sigma_f = \sigma'_f \sqsubseteq \sigma$ (7) - (hyp.4) + (1) + (2)
- $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$ **(8)** - (5) - (7)

[Binary Operation] Suppose that $e = e_0 \ \mathsf{op} \ e_1$ (hyp.5). We conclude that there are two memories μ_0 and μ_0', two labellings Σ_0 and Σ_0', four values v_0, v_1, v_0', and v_1', and four security levels σ_0, σ_1, σ_0', and σ_1' such that:

- $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, v_0, \Sigma_0, \sigma_0 \rangle$, $r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0 \rangle \Downarrow_{IF} \langle \mu_f, v_1, \Sigma_f, \sigma_1 \rangle$, and $v_f = v_0 \ \mathsf{op} \ v_1$, and $\sigma_f = \sigma_0 \sqcup \sigma_1$ (1) - (hyp.2) + (hyp.5)
- $r, \sigma_{pc} \vdash \langle \mu', e_0, \Sigma' \rangle \Downarrow_{IF} \langle \mu_0', v_0', \Sigma_0', \sigma_0' \rangle$, $r, \sigma_{pc} \vdash \langle \mu_0', e_1, \Sigma_0' \rangle \Downarrow_{IF} \langle \mu_f', v_1', \Sigma_f', \sigma_1' \rangle$, $v_f' = v_0' \ \mathsf{op} \ v_1'$, and $\sigma_f' = \sigma_0' \sqcup \sigma_1'$ (2) - (hyp.3) + (hyp.5)
- $\mu_0, \Sigma_0 \sim_\sigma \mu_0', \Sigma_0'$ and $v_0, \sigma_0 \sim_\sigma v_0', \sigma_0'$ (3) - (hyp.1) + (1) + (2) + **ih**
- $\mu_f, \Sigma_f \sim_\sigma \mu_f', \Sigma_f'$ and $v_1, \sigma_1 \sim_\sigma v_1', \sigma_1'$ **(4)** - (1) - (3) + **ih**
- $v_f, \sigma_f \sim_\sigma v_f', \sigma_f'$ **(5)** - (1)-(4)

[Variable] Suppose that $e = x$ (hyp.5). Letting $m_x = \mathsf{string}(x)$, we conclude that there are two references r_x and r_x' such that:

- $\mu_f = \mu$, $\Sigma_f = \Sigma$, $r_x = \mathsf{Scope}(\mu, r, x)$, $v_f = \mu(r_x \cdot m_x)$, and $\sigma_f = \Sigma.\mathsf{val}(r_x \cdot m_x) \sqcup \sigma_{pc}$ (1) - (hyp.2) + (hyp.5)
- $\mu_f' = \mu'$, $\Sigma_f' = \Sigma'$, $r_x' = \mathsf{Scope}(\mu', r, x)$, $v_f' = \mu'(r_x' \cdot m_x)$, $\sigma_f = \Sigma'.\mathsf{val}(r_x' \cdot m_x) \sqcup \sigma_{pc}$ (2) - (hyp.3) + (hyp.5)
- $\mu_f, \Sigma_f \sim_\sigma \mu_f', \Sigma_f'$ **(3)** - (hyp.1) + (1) + (2)
- $\Sigma.\mathsf{struct}(r) \sqcup \Sigma'.\mathsf{struct}(r) \sqsubseteq \sigma_{pc}$ (4) - (hyp.2) + (hyp.3) + Well-Labelled Memory (Lemma A.4)
- $\Sigma.\mathsf{struct}(r) \sqcup \Sigma'.\mathsf{struct}(r) \sqsubseteq \sigma$ (5) - (hyp.4) + (4)
- $r_x = r_x'$ (6) - (hyp.1) + (1) + (2) + (5) + Scope-Chain Indistinguishability (Lemma A.2)
- $\mu(r_x \cdot m_x), \Sigma.\mathsf{val}(r_x \cdot m_x) \sim_\sigma \mu'(r_x \cdot m_x), \Sigma'.\mathsf{val}(r_x \cdot m_x)$ (7) - (hyp.1) + (1) + (2) + (6)
- $v_f, \sigma_f \sim_\sigma v_f', \sigma_f'$ **(8)** - (hyp.4) + (1) + (2) + (7)
- $\mu_f, \Sigma_f \sim_\sigma \mu_f', \Sigma_f'$ **(9)** - (hyp.1) + (1) + (2)

[Variable Assignment] Suppose that $e = x = e$ (hyp.5). Letting $m_x = \mathsf{string}(x)$, we conclude that there are two memories μ_0 and μ_0', two labellings Σ_0 and Σ_0', and two references r_x and r_x', such that:

- $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, v_f, \Sigma_0, \sigma_f \rangle$, $r_x = \mathsf{Scope}(\mu_0, r, x)$, $\mu_f = \mu_0[r_x \cdot x \mapsto v_f]$, and $\Sigma_f = \mathsf{updt}(\Sigma_0, (r_x, x), (\Sigma_0.\mathsf{exist}(r_x \cdot x), \sigma_f))$ (1) - (hyp.2) + (hyp.5)
- $r, \sigma_{pc} \vdash \langle \mu', e, \Sigma' \rangle \Downarrow_{IF} \langle \mu_0', v_f', \Sigma_0, \sigma_f \rangle$, $r_x = \mathsf{Scope}(\mu_0, r, x)$, $\mu_f = \mu_0[r_x \cdot x \mapsto v_f]$, and $\Sigma_f = \mathsf{updt}(\Sigma_0, (r_x, x), (\Sigma_0.\mathsf{exist}(r_x \cdot x), \sigma_f))$ (2) - (hyp.3) + (hyp.5)
- $\mu_0, \Sigma_0 \sim_\sigma \mu_0', \Sigma_0'$ and $v_f, \sigma_f \sim_\sigma v_f', \sigma_f'$ **(3)** - (hyp.1) + (1) + (2) + **ih**
- $\Sigma_0.\mathsf{struct}(r) \sqcup \Sigma_0'.\mathsf{struct}(r) \sqsubseteq \sigma_{pc}$ (4) - (1) + (2) + Well-Labelled Memory (Lemma A.4)
- $r_x = r_x'$ (5) - (hyp.1) + (1) + (2) + (4) + Scope-Chain Indistinguishability (Lemma A.2)
- $\mu_f, \Sigma_f \sim_\sigma \mu_f', \Sigma_f'$ **(6)** - (1) - (3) + (5) + Noninterferent Property Assignment (Proposition A.5)

[Property Look-up] Suppose that $e = e_0[e_1]$ (hyp.5). We conclude that there are two intermediate memories μ_0 and μ_0', two labellings Σ_0 and Σ_0', four references r_0, r_0', $\hat{r}$, and $\hat{r}'$, two strings m_1 and m_1', and six security levels σ_0, σ_1, $\hat{\sigma}$, σ_0', σ_1', and $\hat{\sigma}'$ such that:

- $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, r_0, \Sigma_0, \sigma_0 \rangle$, $r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0 \rangle \Downarrow_{IF} \langle \mu_f, m_1, \Sigma_f, \sigma_1 \rangle$, $\langle \hat{r}, \hat{\sigma} \rangle = \mathsf{Proto}(\mu_f, r_0, m_1, \Sigma_1)$, $\hat{r} = \mathsf{null} \Rightarrow v_f = \mathsf{undefined} \wedge \sigma_f = \sigma_0 \sqcup \sigma_1 \sqcup \hat{\sigma}$, and $\hat{r} \neq \mathsf{null} \Rightarrow v_f = \mu_f(\hat{r} \cdot m_1) \wedge \sigma_f = \sigma_0 \sqcup \sigma_1 \sqcup \hat{\sigma} \sqcup \Sigma.\mathsf{val}(r' \cdot m_1)$ (1) - (hyp.2) + (hyp.5)
- $r, \sigma_{pc} \vdash \langle \mu', e_0, \Sigma' \rangle \Downarrow_{IF} \langle \mu_0', r_0', \Sigma_0', \sigma_0' \rangle$, $r, \sigma_{pc} \vdash \langle \mu_0', e_1, \Sigma_0' \rangle \Downarrow_{IF} \langle \mu_f', m_1', \Sigma_f', \sigma_1' \rangle$, $\langle \hat{r}', \hat{\sigma}' \rangle = \mathsf{Proto}(\mu_f', r_0', m_1', \Sigma_1')$, $\hat{r}' = \mathsf{null} \Rightarrow v_f' = \mathsf{undefined} \wedge \sigma_f' = \sigma_0' \sqcup \sigma_1' \sqcup \hat{\sigma}'$, and $\hat{r}' \neq \mathsf{null} \Rightarrow v_f' = \mu_f'(\hat{r}' \cdot m_1') \wedge \sigma_f' = \sigma_0' \sqcup \sigma_1' \sqcup \hat{\sigma}' \sqcup \Sigma'.\mathsf{val}(\hat{r}' \cdot m_1')$ (2) - (hyp.2) + (hyp.5)

- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0$ and $r_0, \sigma_0 \sim_\sigma r'_0, \sigma'_0$ (3) - (hyp.1) + (1) + (2) + **ih**
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ and $m_1, \sigma_1 \sim_\sigma m'_1, \sigma'_1$ **(4)** - (1) - (3) + **ih**

Supoose that $\sigma_f \sqsubseteq \sigma$ (hyp.6), we conclude that:

- $r_0 = r'_0$, $m_1 = m'_1$, $\sigma_0 = \sigma'_0 \sqsubseteq \sigma$, and $\sigma_1 = \sigma'_1 \sqsubseteq \sigma$ (5) - (hyp.6) + (1)-(4)
- $\hat{\sigma} \sqsubseteq \sigma$ (6) - (hyp.6) + (1)
- $\hat{r} = \hat{r}'$ and $\hat{\sigma} = \hat{\sigma}' \sqsubseteq \sigma$
  (7) - (1) + (2) + (4)-(6) + Prototype-Chain Indistinguishability (Lemma A.3)
- *Suppose:* $\hat{r} \neq \mathsf{null}$ (hyp.7):
  - $\hat{r}' \neq \mathsf{null}$ (8.1) - (hyp.7) + (7)
  - $v_f = \mu_f(\hat{r} \cdot m_1)$ and $\sigma_f = \sigma_0 \sqcup \sigma_1 \sqcup \hat{\sigma} \sqcup \Sigma.\mathsf{val}(r' \cdot m_1)$ (8.2) - (hyp.7) + (1)
  - $v'_f = \mu'_f(\hat{r}' \cdot m'_1)$ and $\sigma'_f = \sigma'_0 \sqcup \sigma'_1 \sqcup \hat{\sigma}' \sqcup \Sigma'.\mathsf{val}(\hat{r}' \cdot m'_1)$ (8.3) - (2) + (8.1)
  - $\Sigma.\mathsf{val}(\hat{r} \cdot m_1) \sqsubseteq \sigma$ (8.4) - (hyp.6) + (8.2)
  - $\mu_f(\hat{r} \cdot m_1) = \mu'_f(\hat{r}' \cdot m'_1)$ and $\Sigma.\mathsf{val}(r' \cdot m_1) = \Sigma'.\mathsf{val}(\hat{r}' \cdot m'_1) \sqsubseteq \sigma$ (8.5) - (4) + (5) + (8.4)
  - $v_f = v'_f$ and $\sigma_f = \sigma'_f \sqsubseteq \sigma$ **(8.6)** - (5) + (7) + (8.2) + (8.3) + (8.5)
- *Suppose:* $\hat{r} = \mathsf{null}$ (hyp.7):
  - $\hat{r}' = \mathsf{null}$ (9.1) - (hyp.7) + (7)
  - $v_f = \mathsf{undefined}$ and $\sigma_f = \sigma_0 \sqcup \sigma_1 \sqcup \hat{\sigma}$ (9.2) - (hyp.7) + (1)
  - $v'_f = \mathsf{undefined}$ and $\sigma'_f = \sigma'_0 \sqcup \sigma'_1 \sqcup \hat{\sigma}'$ (9.3) - (2) + (9.1)
  - $v_f = v'_f$ and $\sigma_f = \sigma'_f \sqsubseteq \sigma$ **(9.4)** - (5) + (7) + (9.2) + (9.3)

[Membership Testing] This case is similar to the previous case. Therefore, the proof is omitted.

[Property Assignment] Suppose that $e = e_0[e_1] = e_2$ (hyp.5). We conclude that there are six intermediate memories μ_0, μ_1, μ_2, μ'_0, μ'_1, and μ'_2, six intermediate labellings Σ_0, Σ_1, Σ_2, Σ'_0, Σ'_1, Σ'_2, two references r_0 and r'_0, two strings m_1 and m'_1, and four security levels σ_0, σ_1, σ'_0, and σ'_1, such that:

- $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, r_0, \Sigma_0, \sigma_0 \rangle$, $r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0 \rangle \Downarrow_{IF} \langle \mu_1, m_1, \Sigma_1, \sigma_1 \rangle$, $r, \sigma_{pc} \vdash \langle \mu_1, e_2, \Sigma_1 \rangle \Downarrow_{IF} \langle \mu_2, v_f, \Sigma_2, \sigma_f \rangle$, $\mu_f = \mu_2[r_0 \cdot m_1 \mapsto v_f]$, and $\Sigma_f = \mathsf{updt}(\Sigma_2, (r_0, m_1), (\sigma_0 \sqcup \sigma_1, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_f))$ (1) - (hyp.2) + (hyp.5)
- $r, \sigma_{pc} \vdash \langle \mu', e_0, \Sigma' \rangle \Downarrow_{IF} \langle \mu'_0, r'_0, \Sigma'_0, \sigma'_0 \rangle$, $r, \sigma_{pc} \vdash \langle \mu'_0, e_1, \Sigma'_0 \rangle \Downarrow_{IF} \langle \mu'_1, m'_1, \Sigma'_1, \sigma'_1 \rangle$, $r, \sigma_{pc} \vdash \langle \mu'_1, e_2, \Sigma'_1 \rangle \Downarrow_{IF} \langle \mu'_2, v'_f, \Sigma'_2, \sigma'_f \rangle$, $\mu'_f = \mu'_2[r'_0 \cdot m'_1 \mapsto v'_f]$, and $\Sigma'_f = \mathsf{updt}(\Sigma'_2, (r'_0, m'_1), (\sigma'_0 \sqcup \sigma'_1, \sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_f))$ (2) - (hyp.3) + (hyp.5)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0$ and $r_0, \sigma_0 \sim_\sigma r'_0, \sigma'_0$ (3) - (hyp.1) + (1) + (2) + **ih**
- $\mu_1, \Sigma_1 \sim_\sigma \mu'_1, \Sigma'_1$ and $m_1, \sigma_1 \sim_\sigma m'_1, \sigma'_1$ (4) - (1) - (3) + **ih**
- $\mu_2, \Sigma_2 \sim_\sigma \mu'_2, \Sigma'_2$ and $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$ **(5)** - (1) + (2) + (4) + **ih**
- Suppose $\sigma_0 \sqcup \sigma_1 \sqsubseteq \sigma$ (hyp.6):
  - $r_0 = r'_0$, $m_1 = m'_1$, $\sigma_0 = \sigma'_0 \sqsubseteq \sigma$, and $\sigma_1 = \sigma'_1 \sqsubseteq \sigma$ (6.1) - (hyp.6) + (3) + (4)
  - $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$
    (6.2) - (1) + (2) + (5) + (6.1) + Noninterferent Property Assignment (Proposition A.5)
- Suppose $\sigma_0 \sqcup \sigma_1 \not\sqsubseteq \sigma$ (hyp.6). This case has four different sub-cases: **(1)** $m_1 \in dom(\mu_2(r_0))$ and $m'_1 \in dom(\mu'_2(r'_0))$, **(2)** $m_1 \in dom(\mu_2(r_0))$ and $m'_1 \notin dom(\mu'_2(r'_0))$, **(3)** $m_1 \notin dom(\mu_2(r_0))$ and $m'_1 \in dom(\mu'_2(r'_0))$, and **(4)** $m_1 \notin dom(\mu_2(r_0))$ and $m'_1 \notin dom(\mu'_2(r'_0))$. We only prove **(2)**, the other cases are equivalent. Hence, suppose that: $m_1 \in dom(\mu_2(r_0))$ (hyp.7) and $m'_1 \notin dom(\mu'_2(r'_0))$ (hyp.8):

- $\sigma_0 \sqcup \sigma_1 \sqsubseteq \Sigma_2.\mathsf{val}(r_0 \cdot m_1)$ (7.1) - (hyp.2) + (hyp.7) + (1)
- $\Sigma_2.\mathsf{val}(r_0 \cdot m_1) \not\sqsubseteq \sigma$ (7.2) - (hyp.6) + (7.1)
- $\mu_2, \Sigma_2 \sim_\sigma \mu_f, \Sigma_f$ (7.3) - (hyp.7) + (1) + (7.2) + Confined Property Assignment (Proposition A.1)
- $\sigma'_0 \sqcup \sigma'_1 \not\sqsubseteq \sigma$ (7.4) - (hyp.6) + (3) + (4)
- $\mu'_2, \Sigma'_2 \sim_\sigma \mu'_f, \Sigma'_f$ (7.5) - (2) + (7.4) + Confined Property Assignment (Proposition A.1)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(7.6)** - (5) + (7.3) + (7.5)

[Property Deletion] This case is similar to the previous case. Therefore, the proof is omitted.

[Function Literal] Suppose that $e = \mathsf{function}^i(x)\{\mathsf{var}\ y_1, \cdots, y_n;\ e\}$ (hyp.5). We conclude that:

- $\mu_f = \mu\,[r_f \mapsto [\texttt{"@fscope"} \mapsto r, \texttt{"@code"} \mapsto \lambda x.\,\{\mathsf{var}\ y_1, \cdots, y_n;\ e\}]]$,
  $\Sigma_f.\mathsf{val} = \Sigma.\mathsf{val}\,[r_f \mapsto [\texttt{"@fscope"} \mapsto \sigma_{pc}, \texttt{"@code"} \mapsto \sigma_{pc}]]$,
  $\Sigma_f.\mathsf{exist} = \Sigma.\mathsf{exist}\,[r_f \mapsto [\texttt{"@fscope"} \mapsto \sigma_{pc}, \texttt{"@code"} \mapsto \sigma_{pc}]]$,
  $\Sigma_f.\mathsf{struct} = \Sigma.\mathsf{struct}\,[r_f \mapsto \sigma_{pc}]$, $v_f = r_f = \mathsf{fresh}(\sigma_{pc})$, and $\sigma_f = \sigma_{pc}$ (1) - (hyp.2) + (hyp.5)
- $\mu'_f = \mu'\left[r'_f \mapsto [\texttt{"@fscope"} \mapsto r, \texttt{"@code"} \mapsto \lambda x.\,\{\mathsf{var}\ y_1, \cdots, y_n;\ e\}]\right]$,
  $\Sigma'_f.\mathsf{val} = \Sigma'.\mathsf{val}\left[r'_f \mapsto [\texttt{"@fscope"} \mapsto \sigma_{pc}, \texttt{"@code"} \mapsto \sigma_{pc}]\right]$,
  $\Sigma'_f.\mathsf{exist} = \Sigma'.\mathsf{exist}\left[r'_f \mapsto [\texttt{"@fscope"} \mapsto \sigma_{pc}, \texttt{"@code"} \mapsto \sigma_{pc}]\right]$,
  $\Sigma'_f.\mathsf{struct} = \Sigma'.\mathsf{struct}\left[r'_f \mapsto \sigma_{pc}\right]$, $v'_f = r'_f = \mathsf{fresh}(\sigma_{pc})$, and $\sigma'_f = \sigma_{pc}$ (2) - (hyp.3) + (hyp.5)
- $r_f = r'_f$ (3) - (hyp.1) + (1) + (2) + *Low-Equal Allocation*
- $v_f = v'_f$ and $\sigma_f = \sigma'_f$ **(4)** - (1) - (3)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(5)** - (hyp.1) + (1) - (3)

[Object Literal] Suppose that $e = \{\,\}^{\sigma_s}$ (hyp.5). We conclude that:

- $\mu_f = \mu\,[r_o \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}]]$,
  $\Sigma_f.\mathsf{val} = \Sigma.\mathsf{val}\,[r_o \mapsto [\texttt{"\_prot\_"} \mapsto \sigma_{pc} \sqcup \sigma_s]]$,
  $\Sigma_f.\mathsf{exist} = \Sigma.\mathsf{exist}\,[r_o \mapsto [\texttt{"\_prot\_"} \mapsto \sigma_{pc} \sqcup \sigma_s]]$,
  $\Sigma_f.\mathsf{struct} = \Sigma.\mathsf{struct}\,[r_o \mapsto \sigma_{pc} \sqcup \sigma_s]$, $v_f = r_o = \mathsf{fresh}(\sigma_{pc})$, and $\sigma_f = \sigma_{pc}$ (1) - (hyp.2) + (hyp.5)
- $\mu'_f = \mu'\,[r'_o \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}]]$,
  $\Sigma'_f.\mathsf{val} = \Sigma'.\mathsf{val}\,[r'_o \mapsto [\texttt{"\_prot\_"} \mapsto \sigma_{pc} \sqcup \sigma_s]]$,
  $\Sigma'_f.\mathsf{exist} = \Sigma'.\mathsf{exist}\,[r'_o \mapsto [\texttt{"\_prot\_"} \mapsto \sigma_{pc} \sqcup \sigma_s]]$,
  $\Sigma'_f.\mathsf{struct} = \Sigma.\mathsf{struct}\,[r_o \mapsto \sigma_{pc} \sqcup \sigma_s]$, $v'_f = r'_o = \mathsf{fresh}(\sigma_{pc})$, and $\sigma'_f = \sigma_{pc}$ (2) - (hyp.3) + (hyp.5)
- $r_o = r'_o$ (3) - (hyp.1) + (1) + (2) + *Low-Equal Allocation*
- $v_f = v'_f$ and $\sigma_f = \sigma'_f$ **(4)** - (1) - (3)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(5)** - (hyp.1) + (1) - (3)

[Function Call] Suppose that $e = e_0(e_1)^i$ (hyp.5). We conclude that there are six intermediate memories μ_0, μ_1, $\hat{\mu}$, μ'_0, μ'_1, and $\hat{\mu}'$, six labellings Σ_0, Σ_1, $\hat{\Sigma}$, Σ'_0, Σ'_1, and $\hat{\Sigma}'$, four references r_0, r_s, r'_0, and r'_s, two values v_2 and v'_2, and four security levels σ_0, σ_1, σ'_0, and σ'_1, such that:

- $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma\rangle \Downarrow_{IF} \langle \mu_0, r_0, \Sigma_0, \sigma_0\rangle$, $r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0\rangle \Downarrow_{IF} \langle \mu_1, v_1, \Sigma_1, \sigma_1\rangle$,
  $\langle \hat{r}, \hat{\mu}, \hat{e}, \hat{\Sigma}\rangle = \mathsf{NewScope}(\mu_1, r_0, v_1, \#glob, \Sigma_1, \sigma_0, \sigma_1)$, and $\hat{r}, \sigma_0 \vdash \langle \hat{\mu}, \hat{e}, \hat{\Sigma}\rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f\rangle$ (1) - (hyp.2) + (hyp.6)

- $r, \sigma_{pc} \vdash \langle \mu', e_0, \Sigma' \rangle \Downarrow_{IF} \langle \mu'_0, r'_0, \Sigma'_0, \sigma'_0 \rangle$, $r, \sigma_{pc} \vdash \langle \mu'_0, e_1, \Sigma'_0 \rangle \Downarrow_{IF} \langle \mu'_1, v'_1, \Sigma'_1, \sigma'_1 \rangle$, $\langle \hat{r}', \hat{\mu}', \hat{e}', \hat{\Sigma}' \rangle = \mathsf{NewScope}(\mu'_1, r'_0, v'_1, \#glob, \Sigma'_1, \sigma'_0, \sigma'_1)$, and $\hat{r}', \hat{\sigma}'_{pc} \vdash \langle \hat{\mu}', \hat{e}', \hat{\Sigma}' \rangle \Downarrow_{IF} \langle \mu'_f, v'_f, \Sigma'_f, \sigma'_f \rangle$ (2) - (hyp.3) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0$ and $r_0, \sigma_0 \sim_\sigma r'_0, \sigma'_0$ (3) - (hyp.1) + (1) + (2) + **ih**
- $\mu_1, \Sigma_1 \sim_\sigma \mu'_1, \Sigma'_1$ and $v_1, \sigma_1 \sim_\sigma v'_1, \sigma'_1$ (4) - (1) - (3) + **ih**

We consider two distinct cases: $\sigma_0 \sqsubseteq \sigma$ and $\sigma_0 \not\sqsubseteq \sigma$. Suppose that $\sigma_0 \sqsubseteq \sigma$ (hyp.6):

- $r_0 = r'_0$ and $\sigma_0 = \sigma'_0 \sqsubseteq \sigma$ (5) - (hyp.6) + (3)
- $\hat{\mu}, \hat{\Sigma} \sim_\sigma \hat{\mu}', \hat{\Sigma}'$ and $\hat{e} = \hat{e}'$ (6) - (1) + (2) + (4) + (5) + Noninterferent Scope Allocation (Proposition A.8)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ and $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$ **(7)** - (1) + (2) + (6) + **ih**

Suppose that $\sigma_0 \not\sqsubseteq \sigma$ (hyp.6):

- $\sigma'_0 \not\sqsubseteq \sigma$ (9) - (hyp.6) + (3)
- $\mu_1, \Sigma_1 \sim_\sigma \hat{\mu}, \hat{\Sigma}$ (10) - (hyp.6) + (1) + Confined Scope Allocation (Proposition A.4)
- $\mu'_1, \Sigma'_1 \sim_\sigma \hat{\mu}', \hat{\Sigma}'$ (11) - (2) + (9) + Confined Scope Allocation (Proposition A.4)
- $\hat{\mu}, \hat{\Sigma} \sim_\sigma \mu_f, \Sigma_f$ (12) - (hyp.6) + (1) + Confinement (Lemma 4.1)
- $\hat{\mu}', \hat{\Sigma}' \sim_\sigma \mu'_f, \Sigma'_f$ (13) - (2) + (9) + Confinement (Lemma 4.1)
- $\mu_1, \Sigma_1 \sim_\sigma \mu_f, \Sigma_f$ (14) - (10) + (12) + Transitivity of $\sim_\sigma$
- $\mu'_1, \Sigma'_1 \sim_\sigma \mu'_f, \Sigma'_f$ (15) - (11) + (13) + Transitivity of $\sim_\sigma$
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(16)** - (4) + (14) + (15) + Symmetry and Reflexivity of $\sim_\sigma$
- $\sigma_f \not\sqsubseteq \sigma$ and $\sigma'_f \not\sqsubseteq \sigma$ **(17)** - (hyp.6) + (1) + (2) + (9) + PC-Conservation (Lemma A.1)

[Method Call] Suppose that $e = e_0[e_1](e_2)^i$ (hyp.5). We conclude that there are eight intermediate memories μ_0, μ_1, μ_2, $\hat{\mu}$, μ'_0, μ'_1, μ'_2, and $\hat{\mu}'$, eight labellings Σ_0, Σ_1, Σ_2, $\hat{\Sigma}$, Σ'_0, Σ'_1, Σ'_2, and $\hat{\Sigma}'$, six references r_0, r_o, r_f, r'_0, r'_o, and r'_f, two strings m_1 and m'_1, two values v_2 and v'_2, and ten security levels σ_0, σ_1, σ_2, σ_o, σ_s, σ'_0, σ'_1, σ'_2, σ'_o, and σ'_s, such that:

- $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, r_0, \Sigma_0, \sigma_0 \rangle$, $r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0 \rangle \Downarrow_{IF} \langle \mu_1, m_1, \Sigma_1, \sigma_1 \rangle$, $r, \sigma_{pc} \vdash \langle \mu_1, e_2, \Sigma_1 \rangle \Downarrow_{IF} \langle \mu_2, v_2, \Sigma_2, \sigma_2 \rangle$, $\langle r_o, \sigma_o \rangle = \mathsf{Proto}(\mu_2, r_0, m_1, \Sigma_2)$, $r_f = \mu_2(r_o \cdot m_1)$, $\sigma_s = \sigma_0 \sqcup \sigma_1 \sqcup \Sigma_2.\mathsf{val}(r_o \cdot m_1) \sqcup \sigma_o$, $\langle \hat{r}, \hat{\mu}, \hat{e}, \hat{\Sigma} \rangle = \mathsf{NewScope}(\mu_2, r_f, v_2, r_0, \Sigma_2, \sigma_s, \sigma_2)$, and $\hat{r}, \hat{\sigma}_{pc} \vdash \langle \hat{\mu}, \hat{e}, \hat{\Sigma} \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$ (1) - (hyp.2) + (hyp.5)
- $r, \sigma_{pc} \vdash \langle \mu', e_0, \Sigma' \rangle \Downarrow_{IF} \langle \mu'_0, r'_0, \Sigma'_0, \sigma'_0 \rangle$, $r, \sigma_{pc} \vdash \langle \mu'_0, e_1, \Sigma'_0 \rangle \Downarrow_{IF} \langle \mu'_1, m'_1, \Sigma'_1, \sigma'_1 \rangle$, $r, \sigma_{pc} \vdash \langle \mu'_1, e_2, \Sigma'_1 \rangle \Downarrow_{IF} \langle \mu'_2, v'_2, \Sigma'_2, \sigma'_2 \rangle$, $\langle r'_o, \sigma'_o \rangle = \mathsf{Proto}(\mu'_2, r'_0, m'_1, \Sigma'_2)$, $r'_f = \mu'_2(r'_o \cdot m'_1)$, $\sigma'_s = \sigma'_0 \sqcup \sigma'_1 \sqcup \Sigma'_2.\mathsf{val}(r'_o \cdot m'_1) \sqcup \sigma'_o$, $\langle \hat{r}', \hat{\mu}', \hat{e}', \hat{\Sigma}' \rangle = \mathsf{NewScope}(\mu'_2, r'_f, v'_2, r'_0, \Sigma'_2, \sigma'_s, \sigma'_2)$, $\hat{r}', \hat{\sigma}'_{pc} \vdash \langle \hat{\mu}', \hat{e}', \hat{\Sigma}' \rangle \Downarrow_{IF} \langle \mu'_f, v'_f, \Sigma'_f, \sigma'_f \rangle$ (2) - (hyp.3) + (hyp.5)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0$ and $r_0, \sigma_0 \sim_\sigma r'_0, \sigma'_0$ (3) - (hyp.1) + (1) + (2) + **ih**
- $\mu_1, \Sigma_1 \sim_\sigma \mu'_1, \Sigma'_1$ and $m_1, \sigma_1 \sim_\sigma m'_1, \sigma'_1$ (4) - (1) - (3) + **ih**
- $\mu_2, \Sigma_2 \sim_\sigma \mu'_2, \Sigma'_2$ and $v_2, \sigma_2 \sim_\sigma v'_2, \sigma'_2$ (5) - (1) + (2) + (4) + **ih**

We consider two distinct cases: either $\sigma_s \sqsubseteq \sigma$ or $\sigma_s \not\sqsubseteq \sigma$. Suppose that $\sigma_s \sqsubseteq \sigma$ (hyp.6), we then conclude that:

- $r_0 = r'_0$ and $\sigma_0 = \sigma'_0 \sqsubseteq \sigma$ (6) - (hyp.6) + (1) + (3)
- $m_1 = m'_1$ and $\sigma_1 = \sigma'_1 \sqsubseteq \sigma$ (7) - (hyp.6) + (1) + (4)

- $r_o = r'_o$ and $\sigma_o = \sigma'_o \sqsubseteq \sigma$
  (8) - (hyp.6) + (1) + (2) + (5) + (6) + (7) + Prototype-Chain Indistinguishability (Lemma A.3)
- $r_f = r'_f$ and $\Sigma_2.\mathsf{val}(r_o \cdot m_1) = \Sigma'_2.\mathsf{val}(r'_o \cdot m'_1) \sqsubseteq \sigma$ (9) - (hyp.6) + (1) + (2) + (5)
- $\sigma_s = \sigma'_s \sqsubseteq \sigma$ (10) - (6) - (9)
- $\hat{\mu}, \hat{\Sigma} \sim_\sigma \hat{\mu}', \hat{\Sigma}'$ and $\hat{e} = \hat{e}'$
  (11) - (1) + (2) + (5) + (10) + Noninterferent Scope Allocation (Proposition A.8)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ and $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$ **(12)** - (1) + (2) + (11) + **ih**

Suppose that $\sigma_s \not\sqsubseteq \sigma$ (hyp.6), we then conclude that:

- $\sigma'_s \not\sqsubseteq \sigma$ (13) - Multiple Steps
- $\mu_2, \Sigma_2 \sim_\sigma \hat{\mu}, \hat{\Sigma}$ (14) - (hyp.6) + (1) + Confined Scope Allocation (Proposition A.4)
- $\mu'_2, \Sigma'_2 \sim_\sigma \hat{\mu}', \hat{\Sigma}'$ (15) - (2) + (13) + Confined Scope Allocation (Proposition A.4)
- $\hat{\mu}, \hat{\Sigma} \sim_\sigma \mu_f, \Sigma_f$ (16) - (hyp.6) + (1) + (14) + Confinement (Lemma 4.1)
- $\hat{\mu}', \hat{\Sigma}' \sim_\sigma \mu'_f, \Sigma'_f$ (17) - (2) + (13) + (15) + Confinement (Lemma 4.1)
- $\mu_2, \Sigma_2 \sim_\sigma \mu_f, \Sigma_f$ (18) - (14) + (16) + Transitivity of $\sim_\sigma$
- $\mu'_2, \Sigma'_2 \sim_\sigma \mu'_f, \Sigma'_f$ (19) - (15) + (17) + Transitivity of $\sim_\sigma$
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(20)** - (5) + (18) + (19) + Symmetry and Reflexivity of $\sim_\sigma$
- $\sigma_f \not\sqsubseteq \sigma$ and $\sigma'_f \not\sqsubseteq \sigma$ **(21)** - (hyp.6) + (1) + (2) + (13) + PC-Conservation (Lemma A.1)

[Sequence] Suppose that $e = e_0, e_1$ (hyp.5). We conclude that there are two memories μ_0 and μ'_0, two labellings Σ_0 and Σ'_0, two values v_0 and v'_0, and two security levels σ_0 and σ'_0 such that:

- $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, v_0, \Sigma_0, \sigma_0 \rangle$ and $r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0 \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$
  (1) - (hyp.2) + (hyp.5)
- $r, \sigma_{pc} \vdash \langle \mu', e_0, \Sigma' \rangle \Downarrow_{IF} \langle \mu'_0, v'_0, \Sigma'_0, \sigma'_0 \rangle$ and $r, \sigma_{pc} \vdash \langle \mu'_0, e_1, \Sigma'_0 \rangle \Downarrow_{IF} \langle \mu'_f, v'_f, \Sigma'_f, \sigma'_f \rangle$
  (2) - (hyp.3) + (hyp.5)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0$ and $v_0, \sigma_0 \sim_\sigma v'_0, \sigma'_0$ (3) - (hyp.1) + (1) + (2) + **ih**
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ and $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$ **(4)** - (1) - (3) + **ih**

[Conditional] Suppose that $e = \hat{e}\ ?\ (e_0) : (e_1)$ (hyp.5). We conclude that there are two memories $\hat{\mu}$ and $\hat{\mu}'$, two labellings $\hat{\Sigma}$ and $\hat{\Sigma}'$, two values $\hat{v}$ and $\hat{v}'$, and two levels $\hat{\sigma}$ and $\hat{\sigma}'$ such that:

- $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \rangle \Downarrow_{IF} \langle \hat{\mu}, \hat{v}, \hat{\Sigma}, \hat{\sigma} \rangle$ and $r, \hat{\sigma} \vdash \langle \hat{\mu}, e_i, \hat{\Sigma} \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$, where $i = 0$ when $\hat{v} \notin \texttt{Falsy}$ and $i = 1$ when $\hat{v} \in \texttt{Falsy}$ (1) - (hyp.2) + (hyp.5)
- $r, \sigma_{pc} \vdash \langle \mu', e, \Sigma' \rangle \Downarrow_{IF} \langle \hat{\mu}', \hat{v}', \hat{\Sigma}', \hat{\sigma}' \rangle$ and $r, \sigma_{pc} \sqcup \hat{\sigma}' \vdash \langle \hat{\mu}', e_j, \hat{\Sigma}' \rangle \Downarrow_{IF} \langle \mu'_f, v'_f, \Sigma'_f, \sigma_f \rangle$, where $j = 0$ if $\hat{v}' \notin \texttt{Falsy}$ and $j = 1$ $\hat{v}' \in \texttt{Falsy}$ (2) - (hyp.3) + (hyp.5)
- $\hat{\mu}, \hat{\Sigma} \sim_\sigma \hat{\mu}', \hat{\Sigma}'$ and $\hat{v}, \hat{\sigma} \sim_\sigma \hat{v}', \hat{\sigma}'$ (3) - (hyp.1) + (1) + (2) + **ih**

Without loss of generality, we assume $i = 0$ (hyp.6) (the case $i = 1$ is symmetric). We proceed by case analysis. Suppose that $\hat{\sigma} \sqsubseteq \sigma$ (hyp.7). We conclude:

- $\hat{v} = \hat{v}'$ and $\hat{\sigma} = \hat{\sigma}' \sqsubseteq \sigma$ (4) - (hyp.7) + (3)
- $\hat{v} = \hat{v}' \notin \texttt{Falsy}$ (5) - (1) + (hyp.6)
- $j = 0$ (6) - (2) + (4) + (5)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ and $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$ **(7)** - (hyp.7) + (1)-(4) + (6) + **ih**

Suppose that $\hat{\sigma} \not\sqsubseteq \sigma$ (hyp.7). We conclude:

- $\hat{\sigma}' \not\sqsubseteq \sigma$ (8) - (hyp.7) + (3)
- $\hat{\mu}, \hat{\Sigma} \sim_\sigma \mu_f, \Sigma_f$ (9) - (hyp.8) + (1) + Confinement (Lemma 4.1)
- $\hat{\mu}', \hat{\Sigma}' \sim_\sigma \mu'_f, \Sigma'_f$ (10) - (2) + (8) + Confinement (Lemma 4.1)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(11)** - (3) + (9) + (10) + Reflexivity and Transitivity of $\sim_\sigma$
- $\sigma_f \sqcap \sigma'_f \not\sqsubseteq \sigma$ **(12)** - (hyp.8) + (1) + (2) + (8) + PC-Conservation (Lemma A.1)

□

A.2 Correctness - Inlining Compiler

In order to prove correctness, one must be able to relate the outcome of applying the prototype-chain and the scope-chain look-up procedures in similar memories. Lemma A.9 states that the results of applying the scope-chain look-up procedure in two similar memories coincide. Analogously, Lemma A.10 states that the results of applying the prototype-chain look-up procedure in two similar memories coincide.

Proposition A.9 (Scope-Chain Similarity)**.** *Given two memories μ and μ' and a labelling Σ such that $\mu, \Sigma \mathcal{S} \mu'$; then, for any reference $r \in \mu$ and identifier x, $r_x = \mathsf{Scope}(\mu, r, x)$ iff $r_x = \mathsf{Scope}(\mu', r, x)$.*

Proof: In order to prove the result, one must prove both sides of the equivalence. The proof of the direct side is follows by induction on the derivation of $r_x = \mathsf{Scope}(\mu, r, x)$, while the proof of the converse side is done by induction on the derivation of $r_x = \mathsf{Scope}(\mu, r, x)$. □

Proposition A.10 (Prototype-Chain Similarity)**.** *Given two memories μ and μ' and a labelling Σ such that $\mu, \Sigma \mathcal{S} \mu'$; then, for any two references reference $r, r' \in dom(\mu)$, property p, and security level σ, $\langle r', \sigma \rangle = \mathsf{Proto}(\mu, r, p, \Sigma)$ iff $r' = \mathsf{Proto}(\mu', r, p)$.*

Proof: In order to prove the result, one must prove both sides of the equivalence. The proof of the direct side is follows by induction on the derivation of $\langle r', \sigma \rangle = \mathsf{Proto}(\mu, r, p, \Sigma)$, while the proof of the converse side is done by induction on the derivation of $r' = \mathsf{Proto}(\mu', r, p)$. □

The following two lemmas state two important properties concerning the prototype-chain and the scope-chain inspection procedures that instrumented memories are proven to verify. Lemma A.6 establishes that the scope object that defines a given variable in a scope-chain is also the scope object that defines its corresponding shadow variable. Analogously, Lemma A.6 establishes that the object that defines a given property in a prototype-chain is also the object that defines its two corresponding shadow properties.

Lemma A.5 (Well-Instrumented Scope-Chain)**.** *For any instrumented memory μ, two references r and r_x, and variable x, it holds that: $r_x = \mathsf{Scope}(\mu, r, x)$ iff $r_x = \mathsf{Scope}(\mu, r, \$x)$.*

Proof: In order to prove the result, one must prove both sides of the equivalence. The proof of the direct side is follows by induction on the derivation of $r_x = \mathsf{Scope}(\mu, r, x)$, while the proof of the converse side is done by induction on the derivation of $r_x = \mathsf{Scope}(\mu, r, \$x)$. □

Lemma A.6 (Well-Instrumented Prototype-Chain)**.** *For any instrumented memory μ, two references r and r_p, and property name p, it holds that: $r_p = \mathsf{Proto}(\mu, r, p)$ iff $r_p = \mathsf{Proto}(\mu, r, \$p)$ iff $r_p = \mathsf{Proto}(\mu, r, \$\bar{p})$.*

Finally, Lemma A.7 states that the value of a bookkeeping variable whose index does not belong to the indexes of the program to compile is not changed by the execution of its respective compilation. In other words, the execution of a compiled program only updates values of bookkeeping variables whose indexes belong to the set of indexes of its original counterpart.

Lemma A.7 (Invariance of Bookkeeping Variables)**.** *For any two instrumented memories μ and μ', scope reference r, expression e, indexes i and j, and value value v, such that $\mathcal{C}\langle e\rangle = \langle \hat{e} \mid j\rangle$, $i \notin indexes(e)$, $\$v_i, \$l_i \in dom(\mu(r))$, and $r \vdash \langle \mu, \hat{e}\rangle \Downarrow \langle \mu', v\rangle$, it holds that: $\mu(r \cdot \$v_i) = \mu'(r \cdot \$v_i)$ and $\mu(r \cdot \$l_i) = \mu'(r \cdot \$l_i)$.*

Theorem 4.3 - Compiler Correctness

Proof: In order to prove the claim, we have to prove both sides of the equivalence. Since the proof is analogous, we choose to prove the right-to-left implication, which immediately implies security. Below, we restate the hypotheses of the theorem:

- $\mu, \Sigma \; \mathcal{S} \; \mu'$ (hyp.1),
- $\mathcal{C}\langle e\rangle = \langle e' \mid i\rangle$ (hyp.2),
- $r \vdash \langle \mu', e'\rangle \Downarrow \langle \mu'_f, v'_f\rangle$ (hyp.3),
- $\sigma_{pc} = \mu'(r \cdot \texttt{"\$pc"})$ (hyp.4)

We have to prove that:

- $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma\rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f\rangle$, for some configuration $\langle \mu_f, v_f, \Sigma_f, \sigma_f\rangle$,
- $\mu_f, \Sigma_f \; \mathcal{S} \; \mu'_f$
- $v_f = v'_f = \mu'_f(r \cdot \$v_i)$,
- $\sigma_f = \mu'_f(r \cdot \$l_i)$,
- $\sigma_{pc} = \mu'_f(r \cdot \texttt{"\$pc"})$

The proof proceeds by induction on the derivation of (hyp.1).

[Value] Suppose that $e = v^i$ (hyp.5). Letting $m_{v_i} = \mathsf{string}(\$v_i)$ and $m_{l_i} = \mathsf{string}(\$l_i)$, we conclude that:

- $e' = \$l_i = \texttt{\$pc},\ \$v_i = v$ (1) - (hyp.2) + (hyp.5)
- $\mu'_f = \mu'[r \cdot m_{l_i} \mapsto \sigma_{pc}, r \cdot m_{v_i} \mapsto v]$ (2) - (hyp.4) + (1)
- $r, \sigma_{pc} \vdash \langle \mu, v, \Sigma\rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f\rangle$ with $\mu_f = \mu$, $\Sigma_f = \Sigma$, $v_f = v$, and $\sigma_f = \sigma_{pc}$ (3) - (hyp.5)
- $\mu_f, \Sigma_f \; \mathcal{S} \; \mu'_f$ **(4)** - (hyp.1) + (2) + (3)
- $\sigma_{pc} = \mu'_f(r \cdot \texttt{"\$pc"})$ **(5)** - (hyp.4) + (3)
- $v_f = v'_f = v = \mu'_f(r \cdot m_{v_i})$ and $\sigma_f = \mu'_f(r \cdot m_{l_i}) = \sigma_{pc}$ **(6)** - (2) + (3)

[This] Suppose that $e = \mathsf{this}^i$ (hyp.5). Letting $m_{v_i} = \mathsf{string}(\$v_i)$ and $m_{l_i} = \mathsf{string}(\$l_i)$, we conclude that:

- $e' = \$l_i = \texttt{\$pc},\ \$v_i = \mathsf{this}$ (1) - (hyp.2) + (hyp.5)
- $\mu'_f = \mu'[r \cdot m_{l_i} \mapsto \sigma_{pc}, r \cdot m_{v_i} \mapsto v'_f]$ and $v'_f = \mu'(r \cdot \texttt{"@this"})$ (2) - (hyp.3) + (1)
- $\mu'_f(r \cdot m_{l_i}) = \sigma_{pc}$ and $\mu'_f(r \cdot m_{v_i}) = \mu'(r \cdot \texttt{"@this"})$ (3) - (hyp.4) + (2)

- $r, \sigma_{pc} \vdash \langle \mu, \mathsf{this}, \Sigma \rangle \Downarrow_{IF} \langle \mu, v_f, \Sigma, \sigma_{pc} \rangle$ and $v_f = \mu(r \cdot \texttt{"@this"})$ **(4)** - (hyp.5)
- $\mu_f, \Sigma_f \mathrel{\mathcal{S}} \mu'_f$ **(5)** - (hyp.1) + (2) + (4)
- $v_f = v'_f = \mu'_f(r \cdot m_{v_i})$ **(6)** - (hyp.1) + (2) + (4)
- $\sigma_{pc} = \mu'_f(r \cdot \texttt{"\$pc"})$ **(7)** - (hyp.4) + (2)
- $\sigma_f = \mu'_f(r \cdot m_{l_i})$ **(8)** - (2) + (4)

[Variable] Suppose that $e = x^i$ (hyp.5). Letting $m_{v_i} = \mathsf{string}(\$v_i)$, $m_{l_i} = \mathsf{string}(\$l_i)$, $m_x = \mathsf{string}(x)$, and $m_{lx} = \mathsf{string}(\$x)$, we conclude that there is a reference r_x such that:

- $e' = \$l_i = \texttt{\$pc} \sqcup \$x,\ \$v_i = x$ (1) - (hyp.2) + (hyp.5)
- $r_x = \mathsf{Scope}(\mu', r, x)$, $r_x \neq \mathsf{null}$, $v'_f = \mu'(r_x \cdot m_x)$, and $\mu'_f = \mu'[r \cdot m_{l_i} \mapsto (\mu'(r_x \cdot m_{lx}) \sqcup \sigma_{pc}), r \cdot m_{v_i} \mapsto v'_f]$ (2) - (hyp.3) + (hyp.4) + (hyp.5) + Well-Instrumented Scope-Chain (Proposition A.6)
- $r_x = \mathsf{Scope}(\mu, r, x)$ (3) - (hyp.1) + (2) + Scope-Chain Similarity (Proposition A.9)
- $r, \sigma_{pc} \vdash \langle \mu, \mathsf{this}, \Sigma \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$ with $\mu_f = \mu$, $\Sigma_f = \Sigma$, $v_f = \mu(r_x \cdot m_x)$, and $\sigma_f = \sigma_{pc} \sqcup \Sigma.\mathsf{val}(r_x \cdot m_x)$ **(4)** - (hyp.5) + (3)
- $\mu_f, \Sigma_f \mathrel{\mathcal{S}} \mu'_f$ **(5)** - (hyp.1) + (2) + (4)
- $v_f = v'_f = \mu'_f(r \cdot m_{v_i})$ **(6)** - (hyp.1) + (2) + (4)
- $\sigma_{pc} = \mu'_f(r \cdot \texttt{"\$pc"})$ **(7)** - (hyp.4) + (2)
- $\sigma_f = \mu'_f(r \cdot m_{l_i})$ **(8)** - (hyp.1) + (2) + (4)

[Binary Operation] Suppose that $e_0\ \mathsf{op}^i\ e_1$ (hyp.5). Letting $m_{v_i} = \mathsf{string}(\$v_i)$, $m_{l_i} = \mathsf{string}(\$l_i)$, $m_{v_j} = \mathsf{string}(\$v_j)$, $m_{l_j} = \mathsf{string}(\$l_j)$, $m_{v_k} = \mathsf{string}(\$v_k)$, and $m_{l_k} = \mathsf{string}(\$l_k)$, we conclude that:

- $e' = e'_0,\ e'_1,\ \$l_i = \$l_j \sqcup \$l_k,\ \$v_i = \$v_j\ \mathsf{op}\ \$v_k$, where: $\mathcal{C}\langle e_0 \rangle = \langle e'_0 \mid j \rangle$ and $\mathcal{C}\langle e_1 \rangle = \langle e'_1 \mid k \rangle$ (1) - (hyp.2) + (hyp.5)
- $r' \vdash \langle \mu', e'_0 \rangle \Downarrow \langle \mu'_0, v'_0 \rangle$, $r' \vdash \langle \mu'_0, e'_1 \rangle \Downarrow \langle \mu'_f, v'_1 \rangle$, $v'_f = \mu'_f(r \cdot m_{v_j})\ \mathsf{op}\ \mu'_f(r \cdot m_{v_k})$, $\sigma'_f = \sigma'_0 \sqcup \sigma'_1$, $\mu'_f = \mu'[r \cdot m_{l_i} \mapsto \sigma'_f, r \cdot m_{v_i} \mapsto v'_f]$, where $\sigma'_0 = \mu'_f(r \cdot m_{l_j})$ and $\sigma'_1 = \mu'_f(r \cdot m_{l_k})$ (2) - (hyp.3) + (hyp.5) + (1)
- There is a configuration $\langle \mu_0, v_0, \Sigma_0, \sigma_0 \rangle$ such that:
  - $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, v_0, \Sigma_0, \sigma_0 \rangle$
  - $\mu_0, \Sigma_0 \mathrel{\mathcal{S}} \mu'_0$
  - $v_0 = v'_0 = \mu'_0(r \cdot m_{v_j})$,
  - $\sigma_0 = \mu'_0(r \cdot m_{l_j})$,
  - $\sigma_{pc} = \mu'_0(r \cdot \texttt{"\$pc"})$

  (3) - (hyp.1) + (hyp.4) + (1) + (2) + **ih**
- There is a configuration $\langle \mu_1, v_1, \Sigma_1, \sigma_1 \rangle$ such that:
  - $r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0 \rangle \Downarrow_{IF} \langle \mu_1, v_1, \Sigma_1, \sigma_1 \rangle$
  - $\mu_1, \Sigma_1 \mathrel{\mathcal{S}} \mu'_f$
  - $v_1 = v'_1 = \mu'_f(r \cdot m_{v_k})$,
  - $\sigma_1 = \mu'_f(r \cdot m_{l_k})$,
  - $\sigma_{pc} = \mu'_f(r \cdot \texttt{"\$pc"})$

  (4) - (1) + (2) + (3) + **ih**
- $v_0 = v'_0 = \mu'_f(r \cdot m_{v_j})$ and $\sigma_0 = \mu'_f(r \cdot m_{l_j})$ (5) - (3) + (4) + Invariance of Bookkeeping Variables (Lemma A.7)

- There is a configuration $\langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$ such that $r, \sigma_{pc} \vdash \langle \mu, e_0 \ \mathsf{op} \ e_1, \Sigma \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$, where $\mu_f = \mu_1$, $\Sigma_f = \Sigma_1$, $v_f = v_0 \ \mathsf{op} \ v_1$, and $\sigma_f = \sigma_0 \sqcup \sigma_1$ **(6)** - (hyp.5) + (3) + (4)
- $\mu_f, \Sigma_f \ \mathcal{S} \ \mu'_f$ and $\sigma_{pc} = \mu'_f(r \cdot \texttt{"\$pc"})$ **(7)** - (4) + (6)
- $v_f = v'_f = \mu'_f(r \cdot m_{v_i})$ and $\sigma_f = \sigma'_f = \mu'_f(r \cdot m_{l_i})$ **(8)** - (2) + (4) + (5)

[Variable Assignment] Suppose that $x = e_0$ (hyp.5). Letting $m_{v_i} = \mathsf{string}(\$v_i)$, $m_{l_i} = \mathsf{string}(\$l_i)$, $m_x = \mathsf{string}(x)$, and $m_{lx} = \mathsf{string}(\$x)$, we conclude that there is a ref. r_x s.t.:

- $e' = e'_0,\ \texttt{\$check}(\texttt{\$pc} \sqsubseteq \$x),\ \$x = \$l_i,\ x = \$v_i$, where: $x \notin \mathtt{S}_{comp}$ and $\mathcal{C}\langle e_0 \rangle = \langle e'_0 \mid i \rangle$ (1) - (hyp.2) + (hyp.5)
- $r \vdash \langle \mu, e'_0 \rangle \Downarrow \langle \mu'_0, v'_0 \rangle$, $r_x = \mathsf{Scope}(\mu'_0, r, x)$, $r_x \neq \mathsf{null}$, $\mu'_f = \mu'[r_x \cdot m_x \mapsto v'_f, r_x \cdot m_{lx} \mapsto \sigma'_f]$, $\mu'_0(r \cdot \texttt{"\$pc"}) \sqsubseteq \mu'_0(r_x \cdot m_{lx})$, where: $v'_f = \mu'_0(r \cdot m_{v_i})$ and $\sigma'_f = \mu'_0(r \cdot m_{l_i})$ (2) - (hyp.3) + (1) + Well-Instrumented Scope-Chain (Proposition A.6)
- There is a configuration $\langle \mu_0, v_0, \Sigma_0, \sigma_0 \rangle$ such that:
  - $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, v_0, \Sigma_0, \sigma_0 \rangle$
  - $\mu_0, \Sigma_0 \ \mathcal{S} \ \mu'_0$
  - $v_0 = v'_f = \mu'_0(r \cdot m_{v_i})$,
  - $\sigma_0 = \sigma'_f = \mu'_0(r \cdot m_{l_i})$,
  - $\sigma_{pc} = \mu'_0(r \cdot \texttt{"\$pc"})$

  (3) - (hyp.1) + (hyp.4) + (1) + (2) + **ih**
- $r_x = \mathsf{Scope}(\mu_0, r, x)$ (4) - (2) + (3) + Scope-Chain Similarity (Proposition A.9)
- $\mu'_0(r \cdot \texttt{"\$pc"}) = \sigma_{pc}$ and $\mu'_0(r_x \cdot m_{lx}) = \Sigma_0.\mathsf{val}(r_x \cdot x)$ (5) - (3) + (4)
- $\sigma_{pc} \sqsubseteq \Sigma_0.\mathsf{val}(r_x \cdot x)$ (6) - (2) + (5)
- There is a configuration $\langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$ such that $r, \sigma_{pc} \vdash \langle \mu, x = e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$, where: $\mu' = \mu_0[r_x \cdot m_x \mapsto v_0]$, $\Sigma_f = \mathsf{updt}(\Sigma_0, (r_x, m_x), (\Sigma_0.\mathsf{exist}(r_x \cdot m_x), \sigma_0))$, $v_f = v_0$, and $\sigma_f = \sigma_0$ **(7)** - (1) + (3) + (6)
- $v_f = v_0 = v'_f = \mu'_f(r \cdot m_{v_i})$ and $\sigma_f = \sigma'_f = \mu'_f(r \cdot m_{l_i})$ **(8)** - (2) + (3) + (7)
- $\mu'_f(r \cdot \texttt{"\$pc"}) = \sigma_{pc}$ **(9)** - (2) + (3)
- $\mu_f, \Sigma_f \ \mathcal{S} \ \mu'_f$ **(10)** - (2) + (3) + (7) + (8)

[Sequence] Suppose that $e_0,\ e_1$ (hyp.5). Letting $m_{v_j} = \mathsf{string}(\$v_j)$, $m_{l_j} = \mathsf{string}(\$l_j)$, $m_{v_k} = \mathsf{string}(\$v_k)$, and $m_{l_k} = \mathsf{string}(\$l_k)$, we conclude that:

- $e' = e'_0,\ e'_1$ where: $\mathcal{C}\langle e_0 \rangle = \langle e'_0 \mid j \rangle$ and $\mathcal{C}\langle e_1 \rangle = \langle e'_1 \mid k \rangle$ (1) - (hyp.2) + (hyp.5)
- $r \vdash \langle \mu', e'_0 \rangle \Downarrow \langle \mu'_0, v'_0 \rangle$ and $r \vdash \langle \mu'_0, e'_1 \rangle \Downarrow \langle \mu'_f, v'_f \rangle$ (2) - (hyp.3) + (hyp.5) + (1)
- There is a configuration $\langle \mu_0, v_0, \Sigma_0, \sigma_0 \rangle$ such that:
  - $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, v_0, \Sigma_0, \sigma_0 \rangle$
  - $\mu_0, \Sigma_0 \ \mathcal{S} \ \mu'_0$
  - $v_0 = v'_0 = \mu'_0(r \cdot m_{v_j})$,
  - $\sigma_0 = \mu'_0(r \cdot m_{l_j})$,
  - $\sigma_{pc} = \mu'_0(r \cdot \texttt{"\$pc"})$

  (3) - (hyp.1) + (hyp.4) + (1) + (2) + **ih**
- There is a configuration $\langle \mu_1, v_1, \Sigma_1, \sigma_1 \rangle$ such that:
  - $r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0 \rangle \Downarrow_{IF} \langle \mu_1, v_1, \Sigma_1, \sigma_1 \rangle$
  - $\mu_1, \Sigma_1 \ \mathcal{S} \ \mu'_f$
  - $v_1 = v'_1 = \mu'_1(r \cdot m_{v_k})$,

  - $\sigma_1 = \mu'_1(r \cdot m_{l_k})$,
  - $\sigma_{pc} = \mu'_0(r \cdot \texttt{"\$pc"})$

  (3) - (hyp.1) + (hyp.4) + (1) + (2) + **ih**

- Letting $\mu_f = \mu_1$, $\Sigma_f = \Sigma_1$, $v_f = v_1$, and $\sigma_f = \sigma_1$, it holds that:
  - $r, \sigma_{pc} \vdash \langle \mu, e_0, e_1, \Sigma \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$
  - $\mu_f, \Sigma_f \; \mathcal{S} \; \mu'_f$
  - $v_f = v'_f = \mu'_f(r \cdot m_{v_k})$,
  - $\sigma_f = \mu'_f(r \cdot m_{l_k})$,
  - $\sigma_{pc} = \mu'_f(r \cdot \texttt{"\$pc"})$

  (4) - (2) + (3)

[Conditional] Suppose that $e = e_0 \; ?^{s,t} \; (e_1) : (e_2)$ (hyp.5). We conclude that:

- The compilation of e is given by:
  $$\hat{e} = \begin{cases} \hat{e}_0,\ \$l_s = \texttt{\$pc},\ \texttt{\$pc} = \texttt{\$pc} \sqcup \$l_i, \\ \$v_i \ ? \\ \quad \big(\hat{e}_1, \$v_t = \$v_j,\ \$l_t = \$l_j\big) \\ \quad : \ \big(\hat{e}_2, \$v_t = \$v_k,\ \$l_t = \$l_k\big), \\ \texttt{\$pc} = \$l_s,\ \$v_t \end{cases}$$
  where $\langle e'_0 \mid i \rangle = \mathcal{C}\langle e_0 \rangle$, $\langle e'_1 \mid j \rangle = \mathcal{C}\langle e_1 \rangle$, and and $\langle e'_2 \mid k \rangle = \mathcal{C}\langle e_2 \rangle$. (1) - (hyp.2) + (hyp.5)
- There is a configuration $\langle \mu'_0, v'_0 \rangle$ and a level σ'_0 such that: $r \vdash \langle \mu', e'_0 \rangle \Downarrow \langle \mu'_0, v'_0 \rangle$ and $\mu'_0 = \mu'_0[r \cdot m_{v_i} \mapsto v'_0, r \cdot m_{l_i} \mapsto \sigma'_0]$ (2) - (hyp.3) + (1)
- There is a configuration $\langle \mu_0, v_0, \Sigma_0, \sigma_0 \rangle$ such that:
  - $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \rangle \Downarrow_{IF} \langle \mu_0, v_0, \Sigma_0, \sigma_0 \rangle$
  - $\mu_0, \Sigma_0 \; \mathcal{S} \; \mu'_0$
  - $v_0 = v'_0 = \mu'_0(r \cdot m_{v_i})$,
  - $\sigma_0 = \sigma'_0 = \mu'_0(r \cdot m_{l_i})$,
  - $\sigma_{pc} = \mu'_0(r \cdot \texttt{"\$pc"})$

  (3) - (hyp.1) + (hyp.4) + (1) + (2) + **ih**

There are two cases to consider: either $v'_0 \in \texttt{Falsy}$ or $v'_0 \notin \texttt{Falsy}$. The treatment of these two cases is symmetrical and therefore we only present the case $v'_0 \notin \texttt{Falsy}$ (hyp.6). We conclude that:

- There are two intermediate memories μ''_0 and μ'_1 such that: $\mu''_0 = \mu'_0[r \cdot \$l_s \mapsto \sigma_{pc}, r \cdot \texttt{"\$pc"} \mapsto \sigma'_0]$, $r \vdash \langle \mu''_0, e'_1 \rangle \Downarrow \langle \mu'_1, v'_t \rangle$, $\mu'_f = \mu'_1[r \cdot \$v_t \mapsto v'_1, r \cdot \$l_t \mapsto \sigma'_1, r \cdot \texttt{"\$pc"} \mapsto \sigma_{pc}]$, and $v'_f = v'_1$
  (4) - (hyp.1) + (hyp.3) + (hyp.4) + (1) + (3)
- $v_0 \notin V_f$ (5) - (hyp.6) + (3)
- $\mu_0, \Gamma_0, \Sigma_0 \; \mathcal{S} \; \mu''_0$ (6) - (3) + (4)
- There is a configuration $\langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$ such that:
  - $r, \sigma_{pc} \vdash \langle \mu, e_1, \Sigma_0 \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$
  - $\mu_f, \Sigma_f \; \mathcal{S} \; \mu'_1$
  - $v_f = v'_1 = \mu'_1(r \cdot m_{v_j})$,
  - $\sigma_f = \sigma'_1 = \mu'_1(r \cdot m_{l_j})$,
  - $\sigma_{pc} = \mu'_1(r \cdot \texttt{"\$pc"})$

  (7) - (1) + (3) + (4) + **ih**
- $r, \sigma_{pc} \vdash \langle \mu, e_0 \ ? \ (e_1) : (e_2), \Sigma \rangle \Downarrow_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \rangle$, $\mu_f, \Sigma_f \; \mathcal{S} \; \mu'_f$, $v_f = v'_f = \mu'_f(r \cdot \$v_t)$, $\sigma_f = \mu'_f(r \cdot \$l_t)$, and $\sigma_{pc} = \mu'_f(r \cdot \texttt{"\$pc"})$ **(8)** - (4) + (7)

The remaining cases are similar. □

Appendix B

Proofs of Chapter 5

B.1 Soundness of the Static Type System

This section presents the proof of Theorem 5.1. This proof is preceded by a series of lemmas that establish useful properties of both the low-equality relation and typable programs.

B.1.1 Properties of Well-Typed Memories

Consider a given memory μ well-typed by a given type-based labelling Σ. Furthermore, consider an object o pointed to by a reference r, which has access to a given property p through its prototype-chain. Let r' be the reference of the object that defines p in the prototype-chain of o and $\dot{\tau}$ and $\dot{\tau}'$ the security types of o and of the object that defines p. Lemma B.1 states that $\dot{\tau}$ and $\dot{\tau}'$ associate the same security type and the same existence level with property p.

Lemma B.1 (Well-Typed Prototype Chains)**.** *Given a memory μ well-typed by Σ, a reference r, and property p, such that $r' = \mathsf{Proto}(\mu, r, p)$,* **then** $\Cap (\Sigma(r), p) = \Cap (\Sigma(r'), p)$*, whenever* $\Cap (\Sigma(r'), p)$ *is defined.*

Proof: We have to prove that given that:

- μ is well-typed by Σ (hyp.1)
- $r' = \mathsf{Proto}(\mu, r, p)$ (hyp.2)
- $\Cap (\Sigma(r'), p) = (\sigma, \dot{\tau})$ is defined (hyp.3)

then, it holds that: $\Cap (\Sigma(r), p) = \Cap (\Sigma(r'), p) = (\sigma, \dot{\tau})$. We proceed by induction on the derivation of (hyp.2).

[Base] $p \in dom(\mu(r))$ (hyp.4). We conclude that:

- $r = r'$ (1) - (hyp.2) + (hyp.4)
- $\Cap (\Sigma(r), p) = (\sigma, \dot{\tau})$ (2) - (hyp.3) + (1)

[Look-up] $p \notin dom(\mu(r))$ (hyp.4). We conclude that:

- $r' = \mathsf{Proto}(\mu, r'', p)$ and $\mu(r \cdot \texttt{"\_prot\_"}) = r''$. (1) - (hyp.2) + (hyp.4)
- $\Cap (\Sigma(r''), p) = \Cap (\Sigma(r'), p) = (\sigma, \dot{\tau})$ (2) - (hyp.1) + (1) + **ih**
- $\Sigma(r'') \preceq \pi_{\texttt{type}}(\Cap (\Sigma(r), \texttt{"\_prot\_"}))$ (3) - (hyp.1) + (1)
- $\lfloor \pi_{\texttt{type}}(\Cap (\Sigma(r), \texttt{"\_prot\_"})) \rfloor \equiv \lfloor \Sigma(r'') \rfloor$ (4) - (3)
- $\Cap (\pi_{\texttt{type}}(\Cap (\Sigma(r), \texttt{"\_prot\_"})), p) = \Cap (\Sigma(r''), p)$ (5) - (4)
- $\Cap (\Sigma(r), p) = \Cap (\pi_{\texttt{type}}(\Cap (\Sigma(r), \texttt{"\_prot\_"})), p)$ (6) - Consistent Prototype
- $\Cap (\Sigma(r), p) = (\sigma, \dot{\tau})$ (7) - (hyp.3) + (2) + (5) + (6)

□

B.1.2 Properties of Low-Equal Memories

We now list some properties of the low-equality definition given in Section 5.2, which are later used in the proofs of soundness of both type systems presented in the chapter.

B.1.2.1 Prototype-Chain Indistinguishability

Suppose a reference r is visible in two *low-equal* memories μ_0 and μ_1, respectively well-typed by Σ_0 and Σ_1. Furthermore, suppose that the object type $\Sigma_0(r)$ associates the property p with a visible existence level and that p is defined in the prototype-chain of the object $\mu_0(r)$. In this scenario, Lemma B.2 states that p is also defined in the prototype-chain of $\mu_1(r)$ and that the reference of the object that actually defines p in the prototype-chain of $\mu_0(r_0)$ coincides with the reference of the object that defines p in the prototype-chain of $\mu_1(r_1)$.

Lemma B.2 (Prototype-Chain Indistinguishability)**.** *For any two memories μ_0 and μ_1 respectively well-typed by Σ_0 and Σ_1, reference r, and property p such that $r_0 = \mathsf{Proto}(\mu_0, r, p)$, $r_1 = \mathsf{Proto}(\mu_1, r, p)$, $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$, and $\pi_{\mathtt{lev}}(\Gamma^{\uparrow}\ (\Sigma_0(r), p)) \sqcup lev(\Sigma_0(r)) \sqsubseteq \sigma$, it holds that: $r_0 = r_1$.*

Proof: We have to prove that given that:

- μ_0 and μ_1 are well-typed by Σ_0 and Σ_1 respectively (hyp.1)
- $r_0 = \mathsf{Proto}(\mu_0, r, p)$ (hyp.2)
- $r_1 = \mathsf{Proto}(\mu_1, r, p)$ (hyp.3)
- $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$ (hyp.4)
- $\pi_{\mathtt{lev}}(\Gamma^{\uparrow}\ (\Sigma_0(r), p)) \sqcup lev(\Sigma_0(r)) \sqsubseteq \sigma$ (hyp.5)

then, it holds that: $r_0 = r_1$. We proceed by induction on the derivation of (hyp.2).

[Null] $r = \mathsf{null}$ (hyp.6). We conclude that:

- $r_0 = r_1 = \mathsf{null}$ — **(1)** - (hyp.2) + (hyp.3) + (hyp.6)

[Base] $p \in dom(\mu_0(r))$ (hyp.6). We conclude that:

- $r_0 = r$ — (1) - (hyp.2) + (hyp.6)
- $\Sigma_0(r) = \Sigma_1(r)$ — (2) - (hyp.4) + (hyp.6)
- $\pi_{\mathtt{lev}}(\Gamma^{\uparrow}\ (\Sigma_0(r), p)) = \pi_{\mathtt{lev}}(\Gamma^{\uparrow}\ (\Sigma_1(r), p)) \sqsubseteq \sigma$ — (3) - (hyp.5) + (2)
- $p \in dom(\mu_1(r))$ — (4) - (hyp.4) + (hyp.6) + (3)
- $r_1 = r$ — (5) - (hyp.3) + (4)
- $r_0 = r_1$ — **(6)** - (1) + (5)

[Look-up] $p \notin dom(\mu_0(r))$ (hyp.6) and $r_0 = \mathsf{Proto}(\mu_0, r'_0, p)$ (hyp.7) where: $r'_0 = \mu_0(r \cdot \mathtt{"\_prot\_"})$ (hyp.8). We conclude that:

- $\Sigma_0(r) = \Sigma_1(r)$, $\pi_{\mathtt{lev}}(\Gamma^{\uparrow}\ (\Sigma_0(r), p)) = \pi_{\mathtt{lev}}(\Gamma^{\uparrow}\ (\Sigma_1(r), p)) \sqsubseteq \sigma$, and $lev(\Sigma_0(r)) = lev(\Sigma_1(r)) \sqsubseteq \sigma$ — (1) - (hyp.4) + (hyp.5)
- $p \notin dom(\mu_1(r))$ — (2) - (hyp.4) + (hyp.6) + (1)
- $r_1 = \mathsf{Proto}(\mu_1, r'_1, p)$, where: $r'_1 = \mu_1(r \cdot \mathtt{"\_prot\_"})$ — (4) - (hyp.2) + (3)
- $\pi_{\mathtt{type}}(\Gamma^{\uparrow}\ (\Sigma_i(r), \mathtt{"\_prot\_"})) \preceq_{proto} \Sigma_i(r)$ for $i = 0, 1$ — (5) - (hyp.1) + (hyp.8) + (4)

- $lev(\pi_{\texttt{type}}(\Sigma_i(r), \texttt{"\_prot\_"})) \sqsubseteq lev(\Sigma_i(r)) \sqsubseteq \sigma$, for $i = 0, 1$ (6) - (1) + (5) + Syntax of Types
- $r'_0 = r'_1$ (7) - (hyp.4) + (hyp.8) + (1) + (6)
- $\Sigma_i(r'_i) \preceq \pi_{\texttt{type}}(\Sigma_i(r), \texttt{"\_prot\_"})$, for $i = 0, 1$ (8) - (hyp.1) + (hyp.8) + (5)
- $\pi_{\texttt{lev}}(\Sigma_i(r'_i), p)) \sqcup lev(\Sigma_i(r'_i)) \sqsubseteq \sigma$, for $i = 0, 1$ (9) - (5) + (8)
- $r_0 = r_1$ **(10)** - (hyp.4) + (hyp.5) +(hyp.7) + (4) + (7) + (9) + **ih**

□

B.1.2.2 Confined Memory Updates

The following two lemmas state two simple confinement properties for memory updates. Lemma B.3 states that the scope-chain obtained by updating the value of a variable that is associated with a *high* security type is low-equal to the original one. Analogously, Lemma B.4 states that the memory obtained by updating the property of a given object associated with a *high* existence level is low-equal to the original one.

Lemma B.3 (Confined Variable Assignment)**.** *For any two memories μ and μ', typing environment Γ, reference r, security level σ, variable x, and value v such that:*

- $\mu' = \mu[r_x \cdot p_x \mapsto v]$ *where:* $p_x = \mathsf{string}(x)$ *and* $r_x = \mathsf{Scope}(\mu, r, x)$,
- $lev(\Gamma(x)) \not\sqsubseteq \sigma$*;*

It holds that: $\Gamma, r \Vdash \mu \sim_\sigma \mu'$.

Proof: Immediate from Definition 5.4. □

Lemma B.4 (Confined Property Assignment)**.** *For any two memories μ and μ', type-based labelling Σ, reference r, security level σ, property name p, and value v such that:*

- $\mu' = \mu[r \cdot p \mapsto v]$,
- $lev(\Sigma.\mathsf{exist}(r \cdot p)) \not\sqsubseteq \sigma$*;*

It holds that: $\Gamma, r \Vdash \mu \sim_\sigma \mu'$.

Proof: Immediate from Definition 5.5. □

B.1.2.3 Indistinguishable Memory Updates

The following two lemmas characterise in which conditions two low-equal memories (according to Definition 5.5) can be updated in a way that preserves the low-equality relation. Lemma B.5 states that the assignment of two low-equal values to the same variable in two low-equal scope-chains yields two low-equal scope-chains. Lemma B.6 states that if one assigns two low-equal values to the same property of two objects pointed to by the same reference in two low-equal memories, the resulting memories are still low-equal.

Lemma B.5 (Indistinguishable Variable Assignment)**.** *For any four memories μ_0, μ_1, μ'_0, and μ'_1, typing environment Γ, reference r, security level σ, variable x, and values v_0 and v_1 such that:*

- $\Gamma, r \Vdash \mu_0 \sim_\sigma \mu_1$,

- $\mu'_0 = \mu_0[r_x \cdot p_x \mapsto v_0]$ *where:* $r_x = \mathsf{Scope}(\mu_0, r, x)$ *and* $p_x = \mathsf{string}(x)$,
- $\mu'_1 = \mu_1[r'_x \cdot p_x \mapsto v_1]$ *where:* $r'_x = \mathsf{Scope}(\mu_1, r, x)$ *and* $p_x = \mathsf{string}(x)$,
- $lev(\Gamma(x)) \sqsubseteq \sigma \Rightarrow v_0 = v_1 \ \wedge \ r_x = r'_x$;

It holds that: $\Gamma, r \Vdash \mu'_0 \sim_\sigma \mu'_1$.

Proof: Immediate from Definition 5.4. □

Lemma B.6 (Indistinguishable Property Assignment)**.** *For any four memories* μ_0, μ_1, μ'_0, *and* μ'_1, *labellings* Σ_0 *and* Σ_1, *reference* r, *string* p, *security level* σ, *and values* v_0 *and* v_1 *such that:*

- $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$,
- $\mu'_0 = \mu_0[r \cdot p \mapsto v_0]$,
- $\mu'_1 = \mu_1[r \cdot p \mapsto v_1]$,
- $lev(\Sigma_0(r)) \sqcup lev(\pi_{\mathit{type}}(\Uparrow (\Sigma_0(r), p))) \sqsubseteq \sigma \Rightarrow v_0 = v_1$;

It holds that: $\mu'_0, \Sigma_0 \sim_\sigma \mu'_1, \Sigma_1$.

Proof: Immediate from Definition 5.5. □

B.1.3 Main Properties of the Static Type System

This section presents the proofs of the the results given in Section 5.3.1. These results correspond to the following properties of the static type system:

- Well-Labeling Preservation - Lemma 5.1
- Confinement - Lemma 5.2
- Soundness (Noninterference) - Theorem 5.1

Lemma 5.1 - Well-Labelling Preservation

Proof: We have to prove that given that:

- $r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu', \Sigma', v \rangle$ (hyp.1)
- $\Gamma, \sigma_{pc} \vdash e : \dot{\tau}$ (hyp.2)
- μ is well-typed by Σ and the current scope-chain is well-typed by Γ and Σ (hyp.3)

It holds that:

- μ' is well-typed by Σ' and the current scope-chain after the execution of e is well-typed by Γ and Σ',
- $v \in \mathtt{Ref} \Rightarrow \Sigma'(v) \preceq \dot{\tau}$

The result follows by induction on the derivation of (hyp.1). We distinguish four types of cases:

1. The cases that do not change the memory: [Value], [This], and [Var].
2. The cases that do not directly change the memory: [Binary Operation], [Property Lookup], [Membership Expression], [Conditional Expression], and [Sequence].

3. The cases that directly change the memory either by creating a new property, updating the value of an existing property, or by deleting an existing property: [Variable Assignment], [Property Assignment], and [Property Deletion].

4. The cases that directly change the memory by allocating a new object: [Function Call], [Method Call], and [Object Literal].

We prove one case of each type. The proofs of the remaining cases follow by similar arguments.

[Variable] Suppose $e = x$, for some variable x (hyp.4). We conclude that there is a reference $r_x \in \texttt{Ref}$ such that:

- $\mu' = \mu$, $\Sigma' = \Sigma$, $v = \mu(r_x \cdot m_x)$, where: $r_x = \mathsf{Scope}(\mu, r, x)$ and $m_x = \mathsf{string}(x)$ (1) - (hyp.1) + (hyp.4)
- $\mu(r_x \cdot m_x) \in \texttt{Ref} \Rightarrow \Sigma(\mu(r_x \cdot m_x)) \preceq \Gamma(x)$ (2) - (hyp.3) + (hyp.4) + (1)
- $\dot{\tau} = \Gamma(x)$ (3) - (hyp.2) + 4
- $v \in \texttt{Ref} \Rightarrow \Sigma'(v) \preceq \dot{\tau}$ **(4)** - (1) - (3)
- μ' is well-typed by Σ' and the current scope-chain is well-typed by Γ and Σ' **(5)** - (hyp.3) + (1)

[Binary Operation] Suppose $e = e_0$ op e_1 for two expressions e_0 and e_1 (hyp.4). We conclude that there is a memory μ_0, type-based labelling Σ_0, values v_0 and v_1, and types $\dot{\tau}_0$ and $\dot{\tau}_1$ such that:

- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, v_0 \rangle$ and $r \vdash \langle \mu_0, \Sigma_0, e_1 \rangle \Downarrow \langle \mu', \Sigma', v_1 \rangle$ where: $v = v_0$ op v_1 (1) - (hyp.1) + (hyp.4)
- $\Gamma, \sigma_{pc} \vdash e_0 : \dot{\tau}_0$ and $\Gamma, \sigma_{pc} \vdash e_1 : \dot{\tau}_1$, where: $\dot{\tau} = \dot{\tau}_0 \curlyvee \dot{\tau}_1$ (2) - (hyp.1) + (hyp.2)
- $v \notin \texttt{Ref}$ (3) - (1)
- μ_0 is well-typed by Σ_0 and the current scope-chain after the execution of e_0 is well-typed by Γ and Σ_0 (4) - (hyp.3) + (1) + (2) + **ih**
- μ' is well-typed by Σ' and the current scope-chain after the execution of e_1 is well-typed by Γ and Σ' **(5)** - (1) + (2) + (4) + **ih**
- $v \in \texttt{Ref} \Rightarrow \Sigma'(v) \preceq \dot{\tau}$ **(6)** - (3)

[Variable Assignment] Suppose $e = x = e_0$ for some variable e and expression e_0 (hyp.4). We conclude that there is a memory μ_0, and a reference r_x such that:

- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma', v \rangle$, $r_x = \mathsf{Scope}(\mu_0, r, x)$, and $\mu' = \mu_0[r_x.m_x \mapsto v]$ where: $m_x = \mathsf{string}(x)$ (1) - (hyp.1) + (hyp.4)
- $\Gamma, \sigma_{pc} \vdash e_0 : \dot{\tau}$ and $\dot{\tau}^{\sigma_{pc}} \preceq \Gamma(x)$ (2) - (hyp.2) + (hyp.4)
- μ_0 is well-typed by Σ', the current scope-chain after the execution of e_0 is well-typed by Γ and Σ', and $v \in \texttt{Ref} \Rightarrow \Sigma'(v) \preceq \dot{\tau}$ **(3)** - (hyp.3) + (1) + (2) + **ih**
- $v \in \texttt{Ref} \Rightarrow \Sigma'(v) \preceq \Gamma(x)$ (4) - (2) + (3)
- $\mu'(r_x \cdot m_x) \in \texttt{Ref} \Rightarrow \Sigma'(\mu(r_x \cdot m_x)) \preceq \Gamma(x)$ (5) - (1) + (5)
- μ' is well-typed by Σ', the current scope-chain after the execution of e is well-typed by Γ and Σ', **(6)** - (3) + (5)

[Object Literal] Suppose $e = \{\,\}^{\dot{\tau}'}$ (hyp.4). We conclude that there is a reference $\hat{r}$ such that:

- $v = \hat{r} = \mathsf{fresh}(lev(\dot{\tau}'))$, $\mu' = \mu\,[\hat{r} \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}]]$, and $\Sigma' = \Sigma\,[\hat{r} \mapsto \dot{\tau}']$ (1) - (hyp.1) + (hyp.4)

- $\dot{\tau}' = \dot{\tau}$ and $\sigma_{pc} \sqsubseteq lev(\dot{\tau}')$ (2) - (hyp.2) + (hyp.4)
- μ' is well-typed by Σ' and the current scope-chain after the execution of e is well-typed by Γ and Σ' **(3)** - (hyp.3) + (1)
- $\Sigma'(v) = \dot{\tau}$ **(4)** - (1) + (2)

□

Lemma 5.2 - Confinement - Static Type System

Proof: We have to prove that given that:

- $r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu', \Sigma', v \rangle$ (hyp.1)
- $\Gamma, \sigma_{pc} \vdash e : \dot{\tau}$ (hyp.2)
- μ is well-typed by Σ and the current scope-chain is well-typed by Γ and Σ (hyp.3)
- $\sigma_{pc} \not\sqsubseteq \sigma$ (hyp.4)

It holds that:

- $\mu \upharpoonright^{\Sigma,\sigma} = \mu' \upharpoonright^{\Sigma',\sigma}$
- $(\mu, r) \upharpoonright^{\Gamma,\sigma} = (\mu', r) \upharpoonright^{\Gamma,\sigma}$.

The result follows by induction on the derivation of (hyp.4). As in the previous lemma, we distinguish four types of cases:

1. The cases that do not change the memory: [Value], [This], and [Var].
2. The cases that do not directly change the memory: [Binary Operation], [Property Lookup], [Membership Expression], [Conditional Expression], and [Sequence].
3. The cases that directly change the memory either by creating a new property, updating the value of an existing property, or by deleting an existing property: [Variable Assignment], [Property Assignment], and [Property Deletion].
4. The cases that directly change the memory by allocating a new object: [Function Call], [Method Call], and [Object Literal].

We prove one case of each type. The proofs of the remaining cases follow by similar arguments.

[Variable] Suppose $e = x$, for some variable x (hyp.5). We conclude that there is a reference $r_x \in \texttt{Ref}$ such that:

- $\mu' = \mu$, $\Sigma' = \Sigma$, (1) - (hyp.1) + (hyp.5)
- $\mu \upharpoonright^{\Sigma,\sigma} = \mu' \upharpoonright^{\Sigma',\sigma}$ and $(\mu, r) \upharpoonright^{\Gamma,\sigma} = (\mu', r) \upharpoonright^{\Gamma,\sigma}$ **(2)** - (1)

[Binary Operation] Suppose $e = e_0 \ \texttt{op} \ e_1$ for two expressions e_0 and e_1 (hyp.5). We conclude that there is a memory μ_0, type-based labelling Σ_0, values v_0 and v_1, and types $\dot{\tau}_0$ and $\dot{\tau}_1$ such that:

- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, v_0 \rangle$ and $r \vdash \langle \mu_0, \Sigma_0, e_1 \rangle \Downarrow \langle \mu', \Sigma', v_1 \rangle$ where: $v = v_0 \ \texttt{op} \ v_1$ (1) - (hyp.1) + (hyp.4)
- $\Gamma, \sigma_{pc} \vdash e_0 : \dot{\tau}_0$ and $\Gamma, \sigma_{pc} \vdash e_1 : \dot{\tau}_1$, where: $\dot{\tau} = \dot{\tau}_0 \curlyvee \dot{\tau}_1$ (2) - (hyp.1) + (hyp.2)
- $\mu \upharpoonright^{\Sigma,\sigma} = \mu_0 \upharpoonright^{\Sigma_0,\sigma}$ and $(\mu, r) \upharpoonright^{\Gamma,\sigma} = (\mu_0, r) \upharpoonright^{\Gamma,\sigma}$ (3) - (hyp.3) + (hyp.4) + (1) + (2) + **ih**

- μ_0 is well-typed by Σ_0 and the current scope-chain is well-typed by Γ and Σ_0 after the evaluation of e_0 (4) - (hyp.3) + (1) + (2) + Well-Labelling Preservation (Lemma 5.1)
- $\mu_0 \restriction^{\Sigma_0,\sigma} = \mu' \restriction^{\Sigma',\sigma}$ and $(\mu_0, r) \restriction^{\Gamma,\sigma} = (\mu', r) \restriction^{\Gamma,\sigma}$ (5) - (hyp.4) + (1) + (2) + (4) + **ih**
- $\mu \restriction^{\Sigma,\sigma} = \mu' \restriction^{\Sigma',\sigma}$ and $(\mu, r) \restriction^{\Gamma,\sigma} = (\mu', r) \restriction^{\Gamma,\sigma}$ **(6)** - (3) + (5)

[Variable Assignment] Suppose $e = x = e_0$ for some variable e and expression e_0 (hyp.5). We conclude that there is a memory μ_0, and a reference r_x such that:

- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma', v \rangle$, $r_x = \mathsf{Scope}(\mu_0, r, x)$, and $\mu' = \mu_0[r_x.m_x \mapsto v]$ where: $m_x = \mathsf{string}(x)$ (1) - (hyp.1) + (hyp.5)
- $\Gamma, \sigma_{pc} \vdash e_0 : \dot{\tau}$ and $\dot{\tau}^{\sigma_{pc}} \preceq \Gamma(x)$ (2) - (hyp.2) + (hyp.5)
- μ_0 is well-typed by Σ' and the current scope-chain after the execution of e_0 is well-typed by Γ and Σ' (3) - (hyp.3) + (1) + (2) + Well-Labelling Preservation (Lemma 5.1)
- $\mu \restriction^{\Sigma,\sigma} = \mu_0 \restriction^{\Sigma_0,\sigma}$ and $(\mu, r) \restriction^{\Gamma,\sigma} = (\mu_0, r) \restriction^{\Gamma,\sigma}$ (4) - (hyp.3) + (hyp.4) + (1) + (2) + **ih**
- $lev(\Gamma(x)) \not\sqsubseteq \sigma$ (5) - (hyp.4) + (2)
- $(\mu', r) \restriction^{\Gamma,\sigma} = (\mu_0, r) \restriction^{\Gamma,\sigma}$ **(6)** - (1) + (5) + Confined Variable Assignment (Lemma B.3)
- $\mu' \restriction^{\Sigma',\sigma} = \mu \restriction^{\Sigma,\sigma}$ **(7)** - (1) + (4)

[Object Literal] Suppose $e = \{\,\}^{\dot{\tau}'}$ (hyp.5). We conclude that there is a reference $\hat{r}$ such that:

- $v = \hat{r} = \mathsf{fresh}(lev(\dot{\tau}'))$, $\mu' = \mu\,[\hat{r} \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}]]$, and $\Sigma' = \Sigma\,[\hat{r} \mapsto \dot{\tau}']$ (1) - (hyp.1) + (hyp.5)
- $\dot{\tau}' = \dot{\tau}$ and $\sigma_{pc} \sqsubseteq lev(\dot{\tau}')$ (2) - (hyp.2) + (hyp.5)
- $lev(\dot{\tau}') = lev(\Sigma'(\hat{r})) \not\sqsubseteq \sigma$ (3) - (hyp.4) + (1) + (2)
- $\mu' \restriction^{\Sigma',\sigma} = \mu \restriction^{\Sigma,\sigma}$ **(4)** - (1) + (3)

□

Theorem 5.1 - Noninterference - Static Type System

Proof: We have to prove that given that:

- $\Gamma, \sigma_{pc} \vdash e : \dot{\tau}$ (hyp.1)
- $r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu_f, \Sigma_f, v_f \rangle$ (hyp.2)
- $r \vdash \langle \mu', \Sigma', e \rangle \Downarrow \langle \mu'_f, \Sigma'_f, v'_f \rangle$ (hyp.3)
- $\mu, \Sigma \sim_\sigma \mu', \Sigma'$ (hyp.4)
- $\Gamma, r \Vdash \mu \sim_\sigma \mu'$ (hyp.5)

It holds that:

1. $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$
2. $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$
3. $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v = v'$

We proceed by induction on the derivation of (hyp.2).

[Val] Suppose $e = v$ for some value v (hyp.6). We conclude that:

- $v_f = v'_f = v$ (1) - (hyp.2) + (hyp.3) + (hyp.6)
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(2)** - (1)
- $\mu_f = \mu$, $\mu'_f = \mu'$, $\Sigma_f = \Sigma$, $\Sigma'_f = \Sigma'$ (3) - (hyp.2) + (hyp.3) + (hyp.6)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(4)** - (hyp.4) + (3)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(5)** - (hyp.5) + (3)

[This] Suppose $e = \mathsf{this}$ (hyp.6). We conclude that:

- $v_f = \mu(r \cdot \texttt{"@this"})$ and $v'_f = \mu'(r \cdot \texttt{"@this"})$ (1) - (hyp.2) + (hyp.3) + (hyp.6)
- $lev(\Gamma(\mathsf{this})) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ (2) - (hyp.5) + (1)
- $\dot{\tau} = \Gamma(\mathsf{this})$ (3) - (hyp.1)
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(4)** - (2) + (3)
- $\mu_f = \mu$, $\mu'_f = \mu'$, $\Sigma_f = \Sigma$, and $\Sigma'_f = \Sigma'$. (5) - (hyp.2) + (hyp.3) + (hyp.6)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(6)** - (hyp.4) + (5)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(7)** - (hyp.5) + (5)

[Variable] Suppose $e = x$, for some variable x (hyp.6). We conclude that there are two references r_x and r'_x such that:

- $v_f = \mu(r_x \cdot x)$ and $r_x = \mathsf{Scope}(\mu, r, x)$ (1) - (hyp.2) + (hyp.6)
- $v'_f = \mu'(r'_x \cdot x)$ and $r'_x = \mathsf{Scope}(\mu', r, x)$ (2) - (hyp.3) + (hyp.6)
- $lev(\Gamma(x)) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ (3) - (hyp.5) + (1) + (2)
- $\dot{\tau} = \Gamma(x)$ (4) - (hyp.1) + (hyp.6)
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(5)** - (3) + (4)
- $\mu = \mu_f$, $\mu' = \mu'_f$, $\Sigma = \Sigma_f$, and $\Sigma' = \Sigma'_f$. (6) - (hyp.2) + (hyp.3) + (hyp.6)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(7)** - (hyp.4) + (6)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(8)** - (hyp.5) + (6)

[Binary Operation] Suppose $e = e_0 \ \mathsf{op} \ e_1$ for two exprs. e_0 and e_1 (hyp.6). We conclude that there are two memories μ_0 and μ'_0, two labellings Σ_0 and Σ'_0, four primitive values v_0, v_1, v'_0, and v'_1, and two security types $\dot{\tau}_0$ and $\dot{\tau}_1$ such that:

- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, v_0 \rangle$, $r \vdash \langle \mu_0, \Sigma_0, e_1 \rangle \Downarrow \langle \mu_f, \Sigma_f, v_1 \rangle$, and $v_f = v_0 \ \mathsf{op} \ v_1$ (1) - (hyp.2) + (hyp.6)
- $r \vdash \langle \mu', \Sigma', e_0 \rangle \Downarrow \langle \mu'_0, \Sigma'_0, v'_0 \rangle$, $r \vdash \langle \mu'_0, \Sigma'_0, e'_1 \rangle \Downarrow \langle \mu'_f, \Sigma'_f, v'_1 \rangle$, and $v'_f = v'_0 \ \mathsf{op} \ v'_1$ (2) - (hyp.3) + (hyp.6)
- $\Gamma, \sigma_{pc} \vdash e_0 : \dot{\tau}_0$, $\Gamma, \sigma_{pc} \vdash e_1 : \dot{\tau}_1$, and $\dot{\tau} = \dot{\tau}_0 \curlyvee \dot{\tau}_1$ (3) - (hyp.1) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0$, $\Gamma, r \Vdash \mu_0 \sim_\sigma \mu'_0$, $lev(\dot{\tau}_0) \sqsubseteq \sigma \Rightarrow v_0 = v'_0$ (4) - (hyp.4) + (hyp.5) + (1) + (2) + (3) + **ih**
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$, $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$, $lev(\dot{\tau}_1) \sqsubseteq \sigma \Rightarrow v_1 = v'_1$ **(5)** - (1) + (2) + (3) + (4) + **ih**
- $v_0 = v'_0 \wedge v_1 = v'_1 \Rightarrow v_f = v'_f$ (6) - (1) + (2)
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow (lev(\dot{\tau}_0) \sqsubseteq \sigma) \wedge (lev(\dot{\tau}_1) \sqsubseteq \sigma)$ (7) - (3)
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(8)** - (4)-(7)

[Variable Assignment] Suppose $e = x = e_0$ for some variable e and expression e_0 (hyp.6). Let $m_x = \mathsf{string}(x)$, we conclude that there are two memories μ_0 and μ'_0 and two references r_x and r'_x such that:

- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_f, v_f \rangle$, $r_x = \mathsf{Scope}(\mu_0, r, x)$, and $\mu_f = \mu_0[r_x.m_x \mapsto v_f]$ (1) - (hyp.2) + (hyp.6)
- $r \vdash \langle \mu', \Sigma', e_0 \rangle \Downarrow \langle \mu'_0, \Sigma'_f, v'_f \rangle$, $r'_x = \mathsf{Scope}(\mu'_0, r, x)$, and $\mu'_f = \mu'_0[r'_x.m_x \mapsto v'_f]$ (2) - (hyp.3) + (hyp.6)
- $\Gamma, \sigma_{pc} \vdash e_0 : \dot{\tau}$ and $\dot{\tau}^{\sigma_{pc}} \preceq \Gamma(x)$ (3) - (hyp.1) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0, \Gamma, r \Vdash \mu_0 \sim_\sigma \mu'_0$, $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(4)** - (hyp.4) + (hyp.5) + (1) + (2) + (3) + **ih**
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(5)** - (1) + (2) + (4)
- $lev(\Gamma(x)) \sqsubseteq \sigma \Rightarrow lev(\dot{\tau}) \sqsubseteq \sigma$ (6) - (3)
- $lev(\Gamma(x)) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ (7) - (4) + (6)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(8)** - (1) + (2) + (4) + (7) + Indistinguishable Variable Assignment (Lemma B.5)

[Object Literal] Suppose $e = \{\,\}^{\dot{\tau}'}$ (hyp.6) for some security type $\dot{\tau}'$. We conclude that there are two reference $\hat{r}$ and $\hat{r}'$:

- $\dot{\tau}' = \dot{\tau}$ and $\sigma_{pc} \sqsubseteq lev(\dot{\tau})$ (1) - (hyp.1) + (hyp.6)
- $\hat{r} = \mathsf{fresh}(lev(\dot{\tau}))$, $\mu_f = \mu\,[\hat{r} \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}]]$, $\Sigma_f = \Sigma\,[\hat{r} \mapsto \dot{\tau}]$, and $v_f = \hat{r}$ (2) - (hyp.2) + (hyp.6) + (1)
- $\hat{r}' = \mathsf{fresh}(lev(\dot{\tau}))$, $\mu'_f = \mu'\,[\hat{r}' \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}]]$, $\Sigma'_f = \Sigma'\,[\hat{r}' \mapsto \dot{\tau}]$, and $v'_f = \hat{r}'$ (3) - (hyp.3) + (hyp.6) + (1)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(4)** - (hyp.5) + (2) + (3)

We consider two cases: either the program does a visible object allocation ($lev(\dot{\tau}) \sqsubseteq \sigma$) or the program does an invisible object allocation ($lev(\dot{\tau}) \not\sqsubseteq \sigma$). Suppose $lev(\dot{\tau}) \sqsubseteq \sigma$ (hyp.7):

- $\hat{r} = \hat{r}'$ (5) - (hyp.4) + (hyp.7) + (2) + (3)
- $\mu_f \restriction^{\Sigma_f,\sigma} = \mu \restriction^{\Sigma,\sigma} \cup \{(\hat{r}, \dot{\tau})\} \cup \{(\hat{r}, \texttt{"\_prot\_"}, \mathsf{null}), (\hat{r}, \texttt{"\_prot\_"})\}$ (6) - (hyp.7) + (2)
- $\mu'_f \restriction^{\Sigma'_f,\sigma} = \mu' \restriction^{\Sigma',\sigma} \cup \{(\hat{r}, \dot{\tau})\} \cup \{(\hat{r}, \texttt{"\_prot\_"}, \mathsf{null}), (\hat{r}, \texttt{"\_prot\_"})\}$ (7) - (hyp.7) + (3) + (6)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(8)** - (hyp.4) + (6) + (7)
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(9)** - (2) + (3) + (5)

Suppose $lev(\dot{\tau}) \not\sqsubseteq \sigma$ (hyp.7):

- $\mu_f \restriction^{\Sigma_f,\sigma} = \mu \restriction^{\Sigma,\sigma}$ (10) - (hyp.7) + (2)
- $\mu'_f \restriction^{\Sigma'_f,\sigma} = \mu' \restriction^{\Sigma',\sigma}$ (11) - (hyp.7) + (3)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(12)** - (hyp.4) + (10) + (11)
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(13)** - (hyp.7)

[Property Look-up] Suppose $e = e_0[e_1, P]$ for two expressions e_0 and e_1 (hyp.6). It follows there are two memories μ_0 and μ'_0, type-based labellings Σ_0 and Σ'_0, references r_0, r'_0, $\hat{r}$, $\hat{r}'$, strings m_1 and m'_1, and security types $\dot{\tau}_0$ and $\dot{\tau}_1$, such that:

- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, r_0 \rangle$, $r \vdash \langle \mu_0, \Sigma_0, e_1 \rangle \Downarrow \langle \mu_f, \Sigma_f, m_1 \rangle$, $\hat{r} = \mathsf{Proto}(\mu_f, r_0, m_1)$, $\hat{r} \neq \mathsf{null} \Rightarrow v_f = \mu_f(\hat{r} \cdot m_1)$, and $\hat{r} = \mathsf{null} \Rightarrow v_f = \mathsf{undefined}$ (1) - (hyp.2) + (hyp.6)
- $r \vdash \langle \mu', \Sigma', e_0 \rangle \Downarrow \langle \mu'_0, \Sigma'_0, r'_0 \rangle$, $r \vdash \langle \mu'_0, \Sigma'_0, e_1 \rangle \Downarrow \langle \mu'_f, \Sigma'_f, m'_1 \rangle$, $\hat{r}' = \mathsf{Proto}(\mu'_f, r'_0, m'_1)$, $\hat{r}' \neq \mathsf{null} \Rightarrow v'_f = \mu'_f(\hat{r}' \cdot m'_1)$, and $\hat{r}' = \mathsf{null} \Rightarrow v'_f = \mathsf{undefined}$ (2) - (hyp.3) + (hyp.6)

- $\Gamma, \sigma_{pc} \vdash e_0 : \dot{\tau}_0$, $\Gamma, \sigma_{pc} \vdash e_1 : \dot{\tau}_1$, $\pi_{\texttt{type}}(\Rsh_{\uparrow}\ (\dot{\tau}_0, P)) = \dot{\tau}_{lu}$, and $\dot{\tau} = \dot{\tau}_{lu}^{lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1)}$
  (3) - (hyp.1) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu_0', \Sigma_0', \Gamma, r \Vdash \mu_0 \sim_\sigma \mu_0'$, $lev(\dot{\tau}_0) \sqsubseteq \sigma \Rightarrow r_0 = r_0'$
  (4) - (hyp.4) + (hyp.5) + (1) + (2) + (3) + **ih**
- $\mu_f, \Sigma_f \sim_\sigma \mu_f', \Sigma_f', \Gamma, r \Vdash \mu_f \sim_\sigma \mu_f'$, $lev(\dot{\tau}_1) \sqsubseteq \sigma \Rightarrow m_1 = m_1'$
  (5) - (1) + (2) + (3) + (4) + **ih**

It remains to prove that $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v_f'$. Assuming that $lev(\dot{\tau}) \sqsubseteq \sigma$ (hyp.7), it follows that:

- $lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqcup lev(\dot{\tau}_{lu}) \sqsubseteq \sigma$ (6) - (hyp.7) + (3)
- $r_0 = r_0'$ and $m_1 = m_1'$ (7) - (4)-(6)
- $m_1 = m_1' \in P$ (8) - (1) + (2) + (7) + Correct Annotation
- $lev(\pi_{\texttt{type}}(\Rsh\ (\dot{\tau}_0, m_1))) \sqsubseteq lev(\pi_{\texttt{type}}(\Rsh_{\uparrow}\ (\dot{\tau}_0, P))) = lev(\dot{\tau}_{lu})$ (9) - (3) + (8)
- $lev(\pi_{\texttt{type}}(\Rsh\ (\dot{\tau}_0, m_1))) \sqcup lev(\dot{\tau}_0) \sqsubseteq \sigma$ (10) - (6) + (9)
- $\Sigma_f(r_0) = \Sigma_f'(r_0') \preceq \dot{\tau}_0$ (11) - (1) - (3) + (5) - (7) + Well-Labelling Preservation (Lemma 5.1)
- $lev(\Sigma_f(r_0)) = lev(\Sigma_f'(r_0')) \sqsubseteq lev(\dot{\tau}_0)$ (12) - (11)
- $\lfloor \Sigma_f(r_0) \rfloor = \lfloor \Sigma_f'(r_0') \rfloor = \lfloor \dot{\tau}_0 \rfloor$ (13) - (11)
- $\Rsh\ (\dot{\tau}_0, m_1) = \Rsh\ (\Sigma_f(r_0), m_1) = \Rsh\ (\Sigma_f'(r_0'), m_1')$ (14) - (13)
- $lev(\pi_{\texttt{type}}(\Rsh\ (\Sigma_f(r_0), m_1))) = lev(\pi_{\texttt{type}}(\Rsh\ (\Sigma_f'(r_0'), m_1'))) \sqsubseteq \sigma$ (15) - (10) + (14)
- $lev(\pi_{\texttt{type}}(\Rsh\ (\Sigma_f(r_0), m_1))) \sqcup lev(\Sigma_f(r_0)) \sqsubseteq \sigma$ (16) - (12) + (15)
- $\hat{r} = \hat{r}'$ and $\hat{r} \neq \mathsf{null} \Rightarrow lev(\Sigma_f(\hat{r})) = lev(\Sigma_f'(\hat{r}')) \sqsubseteq \sigma$
  (17) - (1) + (2) + (5) + (16) + Prototype-Chain Indistinguishability (Lemma B.2)

We consider two cases: $\hat{r} \neq \mathsf{null}$ or $\hat{r} = \mathsf{null}$. Suppose $\hat{r} \neq \mathsf{null}$ (hyp.8):

- $\hat{r}' \neq \mathsf{null}$ (18) - (hyp.8) + (17)
- $\hat{r} = \hat{r}' \neq \mathsf{null}$ and $lev(\Sigma_f(\hat{r})) = lev(\Sigma_f'(\hat{r}')) \sqsubseteq \sigma$ (19) - (hyp.8) + (17)
- $\Rsh\ (\Sigma_f(r_0), m_1) = \Rsh\ (\Sigma_f(\hat{r}), m_1)$
  (20) - (hyp.8) + (1) + Well-Typed Prototype Chains (Lemma B.1)
- $\Rsh\ (\Sigma_f'(r_0'), m_1) = \Rsh\ (\Sigma_f'(\hat{r}'), m_1)$
  (21) - (2) + (19) + Well-Typed Prototype Chains (Lemma B.1)
- $lev(\pi_{\texttt{type}}(\Rsh\ (\Sigma_f(\hat{r}), m_1))) = lev(\pi_{\texttt{type}}(\Rsh\ (\Sigma_f'(\hat{r}), m_1))) \sqsubseteq \sigma$ (22) - (15) + (20) + (21)
- $v_f = v_f'$ **(23)** - (1) + (2) + (5) + (19) + (22)

Suppose $\hat{r} = \mathsf{null}$ (hyp.8):

- $\hat{r}' = \mathsf{null}$ (24) - (hyp.8) + (17)
- $v_f = v_f' = \mathsf{undefined}$ (25) - (hyp.8) + (1) + (2) + (24)

[Membership Testing] Suppose $e = e_0\ \mathsf{in}^P\ e_1$ for two expressions e_0 and e_1 (hyp.6). It follows that there are two memories μ_0 and μ_0', two type-based labellings Σ_0 and Σ_0', two references r_1 and r_1', two strings m_0 and m_0', two security types $\dot{\tau}_0$ and $\dot{\tau}_1$, and a security level σ' such that:

- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, m_0 \rangle$, $r \vdash \langle \mu_0, \Sigma_0, e_1 \rangle \Downarrow \langle \mu_f, \Sigma_f, r_1 \rangle$, $\hat{r} = \mathsf{Proto}(\mu_f, r_1, m_0)$, $\hat{r} \neq \mathsf{null} \Rightarrow v_f = \mathsf{true}$, and $\hat{r} = \mathsf{null} \Rightarrow v = \mathsf{false}$ (1) - (hyp.2) + (hyp.6)
- $r \vdash \langle \mu', \Sigma', e_0 \rangle \Downarrow \langle \mu_0', \Sigma_0', m_0' \rangle$, $r \vdash \langle \mu_0', \Sigma_0', e_1 \rangle \Downarrow \langle \mu_f,' \Sigma_f', r_1' \rangle$, $\hat{r}' = \mathsf{Proto}(\mu_f', r_1', m_0')$, $\hat{r}' \neq \mathsf{null} \Rightarrow v_f' = \mathsf{true}$, and $\hat{r}' = \mathsf{null} \Rightarrow v_f' = \mathsf{false}$ (2) - (hyp.2) + (hyp.6)

- $\Gamma, \sigma_{pc} \vdash e_0 : \dot{\tau}_0$, $\Gamma, \sigma_{pc} \vdash e_1 : \dot{\tau}_1$, $\sigma' = lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqcup \pi_{\texttt{lev}}(\Rsh_{\uparrow}\ (\dot{\tau}_1, P))$, and $\dot{\tau} = \mathsf{PRIM}^{\sigma'}$
  (3) - (hyp.1) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0, \Gamma, r \Vdash \mu_0 \sim_\sigma \mu'_0$, $lev(\dot{\tau}_0) \sqsubseteq \sigma \Rightarrow m_0 = m'_0$
  (4) - (hyp.4) + (hyp.5) + (1) - (3) + **ih**
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f, \Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$, $lev(\dot{\tau}_1) \sqsubseteq \sigma \Rightarrow r_1 = r'_1$ **(5)** - (1) - (4) + **ih**

It remains to prove that $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$. Assuming that $lev(\dot{\tau}) \sqsubseteq \sigma$ (hyp.7), it follows that:

- $lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqcup \pi_{\texttt{lev}}(\Rsh_{\uparrow}\ (\dot{\tau}_1, P)) \sqsubseteq \sigma$ (6) - (hyp.7) + (3)
- $m_0 = m'_0$ and $r_1 = r'_1$ (7) - (4)-(6)
- $m_0 = m'_0 \in P$ (8) - (1) + (2) + (7) + Correct Annotation
- $\pi_{\texttt{lev}}(\Rsh\ (\dot{\tau}_1, m_0)) \sqsubseteq \pi_{\texttt{lev}}(\Rsh_{\uparrow}\ (\dot{\tau}_1, P)) \sqsubseteq \sigma$ (9) - (6) + (8)
- $\pi_{\texttt{lev}}(\Rsh\ (\dot{\tau}_1, m_0)) \sqsubseteq \sigma$ (10) - (9)
- $\Sigma_f(r_1) \curlyvee \Sigma'_f(r'_1) \preceq \dot{\tau}_1$ (11) - (1) - (3) + Well-Labelling Preservation (Lemma 5.1)
- $\Sigma_f(r_1) = \Sigma'_f(r'_1) \preceq \dot{\tau}_1$ (12) - (5) + (6) + (11)
- $\lfloor \Sigma_f(r_1) \rfloor \equiv \lfloor \dot{\tau}_1 \rfloor$ and $lev(\Sigma_f(r_1)) \sqsubseteq lev(\dot{\tau}_1)$ (13) - (12)
- $\pi_{\texttt{lev}}(\Rsh\ (\Sigma_f(r_1), m_0)) = \pi_{\texttt{lev}}(\Rsh\ (\dot{\tau}_1, m_0)) \sqsubseteq \sigma$ (14) - (10) + (13)
- $lev(\Sigma_f(r_1)) \sqsubseteq \sigma$ (15) - (6) + (11)
- $\pi_{\texttt{lev}}(\Rsh\ (\Sigma_f(r_1), m_0)) \sqcup lev(\Sigma_f(r_1)) \sqsubseteq \sigma$ (16) - (14) + (15)
- $\hat{r} = \hat{r}'$ and $\hat{r} \neq \mathsf{null} \Rightarrow lev(\Sigma_f(\hat{r})) = lev(\Sigma'_f(\hat{r}')) \sqsubseteq \sigma$
  (17) - (1) + (2) + (5) + (7) + (16) + Prototype-Chain Indistinguishability (Lemma B.2)
- $v_f = v'_f$ **(18)** - (1) + (2) + (17)

[Property Assignment] Suppose $e = e_0[e_1] = e_2$ for three expressions e_0, e_1, and e_2 (hyp.6). We conclude that there are six memories μ_0, μ_1, μ_2, μ'_0, μ'_1, and μ'_2, four type-based labellings Σ_0, Σ_1, Σ'_0, Σ'_1, two references r_0 and r'_0, two strings m_1 and m'_1, and three security types $\dot{\tau}_0$, $\dot{\tau}_1$, and $\dot{\tau}_2$, such that:

- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, r_0 \rangle$, $r \vdash \langle \mu_0, \Sigma_0, e_1 \rangle \Downarrow \langle \mu_1, \Sigma_1, m_1 \rangle$, $r \vdash \langle \mu_1, \Sigma_1, e_2 \rangle \Downarrow \langle \mu_2, \Sigma_f, v_f \rangle$, and $\mu_f = \mu_2[r_0 \cdot m_1 \mapsto v_f]$
  (1) - (hyp.2) + (hyp.6)
- $r \vdash \langle \mu', \Sigma', e_0 \rangle \Downarrow \langle \mu'_0, \Sigma'_0, r'_0 \rangle$, $r \vdash \langle \mu'_0, \Sigma'_0, e_1 \rangle \Downarrow \langle \mu'_1, \Sigma'_1, m'_1 \rangle$, $r \vdash \langle \mu'_1, \Sigma'_1, e_2 \rangle \Downarrow \langle \mu'_2, \Sigma'_f, v'_f \rangle$, and $\mu'_f = \mu'_2[r'_0 \cdot m'_1 \mapsto v'_f]$
  (2) - (hyp.3) + (hyp.6)
- $\Gamma, \sigma_{pc} \vdash e_0 : \dot{\tau}_0$, $\Gamma, \sigma_{pc} \vdash e_1 : \dot{\tau}_1$, $\Gamma, \sigma_{pc} \vdash e_2 : \dot{\tau}_2$, $\dot{\tau} = \dot{\tau}_2$, $\dot{\tau}_2 \preceq \pi_{\texttt{type}}(\Rsh_{\downarrow}\ (\dot{\tau}_0, P))$, $\sigma_{pc} \sqcup lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqsubseteq \pi_{\texttt{lev}}(\Rsh_{\downarrow}\ (\dot{\tau}_0, P))$
  (3) - (hyp.1) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0, \Gamma, r \Vdash \mu_0 \sim_\sigma \mu'_0$, $lev(\dot{\tau}_0) \sqsubseteq \sigma \Rightarrow r_0 = r'_0$
  (4) - (hyp.4) + (hyp.5) + (1) - (3) + **ih**
- $\mu_1, \Sigma_1 \sim_\sigma \mu'_1, \Sigma'_1, \Gamma, r \Vdash \mu_1 \sim_\sigma \mu'_1$, $lev(\dot{\tau}_1) \sqsubseteq \sigma \Rightarrow m_1 = m'_1$ (5) - (1) - (4) + **ih**
- $\mu_2, \Sigma_f \sim_\sigma \mu'_2, \Sigma'_f, \Gamma, r \Vdash \mu_2 \sim_\sigma \mu'_2$, $lev(\dot{\tau}_2) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(6)** - (1) - (3) + (5) + **ih**

We distinguish two different cases, either $\sigma_{pc} \sqcup lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqsubseteq \sigma$ or $\sigma_{pc} \sqcup lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \not\sqsubseteq \sigma$. Suppose $\sigma_{pc} \sqcup lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqsubseteq \sigma$ (hyp.7), it follows that:

- $r_0 = r'_0$ and $m_1 = m'_1$ (7) - (hyp.7) + (4) + (5)
- $\Sigma_f(r_0) \curlyvee \Sigma'_f(r_0) \preceq \dot{\tau}_0$ (8) - (1) - (3) + (7) + Well-Labelling Preservation (Lemma 5.1)
- $\Sigma_f(r_0) = \Sigma'_f(r_0) \preceq \dot{\tau}_0$ (9) - (hyp.7) + (6) + (8)
- $\lfloor \Sigma_f(r_0) \rfloor \equiv \lfloor \Sigma'_f(r_0) \rfloor \equiv \lfloor \dot{\tau}_0 \rfloor$ (10) - (9)

- $lev(\Sigma_f(r_0)) = lev(\Sigma'_f(r_0)) \sqsubseteq lev(\dot{\tau}_0) \sqsubseteq \sigma$ (11) - (hyp.7) + (9)
- $m_1 = m'_1 \in P$ (12) - (1) + (2) + (7) + Correct Annotation
- $\pi_{\texttt{type}}(\Gamma_{\downarrow}\ (\dot{\tau}_0, P)) \preceq \pi_{\texttt{type}}(\Gamma\ (\dot{\tau}_0, m_1))$ (13) - (12)
- $\pi_{\texttt{type}}(\Gamma\ (\dot{\tau}_0, m_1)) = \pi_{\texttt{type}}(\Gamma\ (\Sigma_f(r_0), m_1))$ (14) - (10)
- $\dot{\tau}_2 \preceq \pi_{\texttt{type}}(\Gamma\ (\Sigma_f(r_0), m_1))$ (15) - (3) + (13) + (14)
- $lev(\Sigma_f(r_0)) \sqcup lev(\pi_{\texttt{type}}(\Gamma\ (\Sigma_f(r_0), m_1))) \sqsubseteq \sigma \Rightarrow lev(\dot{\tau}_2) \sqsubseteq \sigma$ (16) - (11) + (15)
- $lev(\Sigma_f(r_0)) \sqcup lev(\pi_{\texttt{type}}(\Gamma\ (\Sigma_f(r_0), m_1))) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ (17) - (6) + (16)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(18)** - (1) + (2) + (6) + (17) + Indistinguishable Property Assignment (Lemma B.6)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(19)** - (1) + (2) + (6)

Suppose $lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \not\sqsubseteq \sigma$ (hyp.7), it follows that:

- $\Sigma_f(r_0) \curlyvee \Sigma'_f(r'_0) \preceq \dot{\tau}_0$ (20) - (1) - (3) + Well-Labelling Preservation (Lemma 5.1)
- $\lfloor \Sigma_f(r_0) \rfloor \equiv \lfloor \Sigma'_f(r'_0) \rfloor \equiv \lfloor \dot{\tau}_0 \rfloor$ (21) - (20)
- $\{m_1, m'_1\} \subseteq P$ (22) - (1) + (2) + Correct Annotation
- $\pi_{\texttt{lev}}(\Gamma_{\downarrow}\ (\dot{\tau}_0, P)) \sqsubseteq \pi_{\texttt{lev}}(\Gamma\ (\dot{\tau}_0, m_1)) \sqcap \pi_{\texttt{lev}}(\Gamma\ (\dot{\tau}_0, m'_1))$ (23) - (22)
- $\pi_{\texttt{lev}}(\Gamma\ (\dot{\tau}_0, m_1)) \sqcap \pi_{\texttt{lev}}(\Gamma\ (\dot{\tau}_0, m'_1)) \not\sqsubseteq \sigma$ (24) - (3) + (23)
- $\pi_{\texttt{lev}}(\Gamma\ (\Sigma_f(r_0), m_1)) \sqcap \pi_{\texttt{lev}}(\Gamma\ (\Sigma'_f(r'_0), m'_1)) \not\sqsubseteq \sigma$ (25) - (21) + (24)
- $\mu_2, \Sigma_2 \sim_\sigma \mu_f, \Sigma_f$ (26) - (1) + (25) + Confined Property Assignment (Lemma B.4)
- $\mu'_2, \Sigma'_2 \sim_\sigma \mu'_f, \Sigma'_f$ (27) - (2) + (25) + Confined Property Assignment (Lemma B.4)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(28)** - (6) + (26) + (27)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(29)** - (1) + (2) + (6)

[Function Call] Suppose $e = e_0(e_1)$ for two expressions e_0 and e_1 (hyp.6). We conclude that there are six memories μ_0, μ_1, $\hat{\mu}$, μ'_0, μ'_1, and $\hat{\mu}'$, four type-based labellings Σ_0, Σ_1, Σ'_0, and Σ'_1, four references r_0, $\hat{r}$, r'_0, and $\hat{r}'$, two values v_1 and v'_1, two expressions $\hat{e}$ and $\hat{e}'$, two types $\dot{\tau}_1$ and $\dot{\tau}_2$, and a security level σ', such that:

- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, r_0 \rangle$, $r \vdash \langle \mu_0, \Sigma_0, e_1 \rangle \Downarrow \langle \mu_1, \Sigma_1, v_1 \rangle$, $\hat{r} \vdash \langle \hat{\mu}, \Sigma_1, \hat{e} \rangle \Downarrow \langle \mu_f, \Sigma_f, v_f \rangle$, $\langle \hat{\mu}, \hat{e}, \hat{r} \rangle = \mathsf{NewScope}(\mu_1, r_0, v_1, \#glob, \Sigma_1)$ (1) - (hyp.2) + (hyp.6)
- $r \vdash \langle \mu', \Sigma', e_0 \rangle \Downarrow \langle \mu'_0, \Sigma'_0, r'_0 \rangle$, $r \vdash \langle \mu'_0, \Sigma'_0, e_1 \rangle \Downarrow \langle \mu'_1, \Sigma'_1, v'_1 \rangle$, $\hat{r}' \vdash \langle \hat{\mu}', \Sigma'_1, \hat{e}' \rangle \Downarrow \langle \mu'_f, \Sigma'_f, v'_f \rangle$, $\langle \hat{\mu}', \hat{e}', \hat{r}' \rangle = \mathsf{NewScope}(\mu'_1, r'_0, v'_1, \#glob, \Sigma'_1)$ (2) - (hyp.3) + (hyp.6)
- $\Gamma, \sigma_{pc} \vdash e_0 : \dot{\tau}_0$, $\Gamma, \sigma_{pc} \vdash e_1 : \dot{\tau}_1$, $\dot{\tau}_0 = \langle \dot{\tau}'_0 . \dot{\tau}'_1 \xrightarrow{\hat{\sigma}} \dot{\tau}'_2 \rangle^{\hat{\sigma}'}$, $\sigma' = lev(\dot{\tau}_0) \sqcup \sigma_{pc} \sqsubseteq \hat{\sigma}$, $\dot{\tau}^{\sigma'}_{global} \preceq \dot{\tau}'_0$, $\dot{\tau}_1^{\sigma'} \preceq \dot{\tau}'_1$, and $\dot{\tau} = (\dot{\tau}'_2)^{\sigma'}$ (3) - (hyp.1) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0$, $\Gamma, r \Vdash \mu_0 \sim_\sigma \mu'_0$, $lev(\dot{\tau}_0) \sqsubseteq \sigma \Rightarrow r_0 = r'_0$ (4) - (hyp.4) + (hyp.5) + (1) + (2) + (3) + **ih**
- $\mu_1, \Sigma_1 \sim_\sigma \mu'_1, \Sigma'_1$, $\Gamma, r \Vdash \mu_1 \sim_\sigma \mu'_1$, $lev(\dot{\tau}_1) \sqsubseteq \sigma \Rightarrow m_1 = m'_1$ (5) - (1) + (2) + (3) + (4) + **ih**

We consider two cases: $\sigma' = lev(\dot{\tau}_0) \sqcup \sigma_{pc} \sqsubseteq \sigma$ and $\sigma' \not\sqsubseteq \sigma$. Suppose $\sigma' \sqsubseteq \sigma$ (hyp.7). It follows that:

- $r_0 = r'_0$ (6) - (hyp.7) + (4)
- $\Sigma_1(r_0) \curlyvee \Sigma'_1(r'_0) \preceq \dot{\tau}_0$ (7) - (1) - (3) + Well-Labelling Preservation (Lemma 5.1)
- $\Sigma_1(r_0) = \Sigma'_1(r'_0) \preceq \dot{\tau}_0$ (8) - (hyp.7) + (5) - (7)
- $\lfloor \Sigma_1(r_0) \rfloor \equiv \lfloor \Sigma'_1(r'_0) \rfloor \equiv \lfloor \dot{\tau}_0 \rfloor \equiv \langle \dot{\tau}'_0 . \dot{\tau}'_1 \xrightarrow{\hat{\sigma}} \dot{\tau}'_2 \rangle$ (9) - (8)

- $lev(\Sigma_1(r_0)) = lev(\Sigma_1'(r_0')) \sqsubseteq lev(\dot{\tau}_0) \sqsubseteq \sigma$ (10) - (hyp.7) + (8)
- $\begin{cases} \mu_1(r_0 \cdot \texttt{"@code"}) = \mu_1'(r_0' \cdot \texttt{"@code"}) = \lambda^{\hat{\Gamma},\Sigma_1(r_0)} x.\left\{\mathsf{var}^{\dot{\tau}_{y_1},\cdots,\dot{\tau}_{y_n}}\ y_1,\cdots,y_n;\ \hat{e}\right\} \\ \mu_1(r_0 \cdot \texttt{"@fscope"}) = \mu_1'(r_0' \cdot \texttt{"@fscope"}) = \hat{r} = \hat{r}' \\ \hat{\Gamma}, \hat{r} \Vdash \mu_1 \sim_\sigma \mu_1' \\ \hat{e} = \hat{e}' \end{cases}$
  for some typing environment $\hat{\Gamma}$ and variables $x, y_1, \cdots, y_n$ (11) - (5) + (9) + (10)
- $\bar{\Gamma}, \hat{\sigma} \vdash \hat{e} : \dot{\tau}_2'$, where $\bar{\Gamma} = \hat{\Gamma}\left[\mathsf{this} \mapsto \dot{\tau}_0', x \mapsto \dot{\tau}_1', y_1 \mapsto \dot{\tau}_{y_1}, \cdots, y_n \mapsto \dot{\tau}_{y_n}\right]$
  (12) - (11) + Well-Labelling Preservation (Lemma 5.1)
- $lev(\dot{\tau}_0') \sqsubseteq \sigma \Rightarrow \#glob = \#glob$ (13) - tautology
- $lev(\dot{\tau}_1') \sqsubseteq \sigma \Rightarrow lev(\dot{\tau}_1) \sqsubseteq \sigma$ (14) - (3)
- $lev(\dot{\tau}_1') \sqsubseteq \sigma \Rightarrow v_1 = v_1'$ (15) - (5) + (14)
- $\bar{\Gamma}, \hat{r} \Vdash \hat{\mu} \sim_\sigma \hat{\mu}'$ (16) - (1) + (2) + (5) + (10) - (13) + (15)
- $\hat{\mu}, \Sigma_1 \sim_\sigma \hat{\mu}', \Sigma_1'$ (17) - (1) + (2) + (5)
- $\mu_f, \Sigma_f \sim_\sigma \mu_f', \Sigma_f', \Gamma, r \Vdash \mu_f \sim_\sigma \mu_f', lev(\dot{\tau}_2') \sqsubseteq \sigma \Rightarrow v_f = v_f'$
  (18) - (1) + (2) + (12) + (17) + **ih**
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v_f'$ **(19)** - (3) + (18)

Suppose $lev(\dot{\tau}_0) \not\sqsubseteq \sigma$ (hyp.7). It follows that:

- $\hat{\sigma} \not\sqsubseteq \sigma$ (20) - (hyp.7) + (3)
- $\Sigma_1(r_0) \curlyvee \Sigma_1'(r_0') \preceq \dot{\tau}_0$ (21) - (1) - (3) + Well-Labelling Preservation (Lemma 5.1)
- $\lfloor \Sigma_1(r_0) \rfloor \equiv \lfloor \Sigma_1'(r_0') \rfloor \equiv \lfloor \dot{\tau}_0 \rfloor \equiv \langle \dot{\tau}_0' . \dot{\tau}_1' \xrightarrow{\hat{\sigma}} \dot{\tau}_2' \rangle$ (22) - (21)
- $\begin{cases} \mu_1(r_0 \cdot \texttt{"@code"}) = \lambda^{\hat{\Gamma},\Sigma_1(r_0)} x.\left\{\mathsf{var}^{\dot{\tau}_{y_1},\cdots,\dot{\tau}_{y_n}}\ y_1,\cdots,y_n;\ \hat{e}\right\} \\ \mu_1(r_0 \cdot \texttt{"@fscope"}) = \hat{r} \\ \bar{\Gamma} = \hat{\Gamma}\left[\mathsf{this} \mapsto \dot{\tau}_0', x \mapsto \dot{\tau}_1', y_1 \mapsto \dot{\tau}_{y_1}, \cdots, y_n \mapsto \dot{\tau}_{y_n}\right] \\ \bar{\Gamma}, \hat{\sigma} \vdash \hat{e} : \dot{\tau}_2' \end{cases}$
  (23) - (21) + (22) + Well-Labelling Preservation (Lemma 5.1)
- $\begin{cases} \mu_1'(r_0' \cdot \texttt{"@code"}) = \lambda^{\hat{\Gamma}',\Sigma_1'(r_0')} x'.\left\{\mathsf{var}^{\dot{\tau}_{y_1'},\cdots,\dot{\tau}_{y_k'}}\ y_1',\cdots,y_k';\ \hat{e}'\right\} \\ \mu_1'(r_0' \cdot \texttt{"@fscope"}) = \hat{r}' \\ \bar{\Gamma}' = \hat{\Gamma}'\left[\mathsf{this} \mapsto \dot{\tau}_0', x' \mapsto \dot{\tau}_1', y_1' \mapsto \dot{\tau}_{y_1'}, \cdots, y_n' \mapsto \dot{\tau}_{y_n'}\right] \\ \bar{\Gamma}', \hat{\sigma} \vdash \hat{e}' : \dot{\tau}_2' \end{cases}$
  (24) - (21) + (22) + Well-Labelling Preservation (Lemma 5.1)
- $\hat{\mu} \restriction^{\Sigma_1,\sigma} = \mu_1 \restriction^{\Sigma_1,\sigma}$ and $(\hat{\mu}, r) \restriction^{\Gamma,\sigma} = (\mu_1, r) \restriction^{\Gamma,\sigma}$ (25) - (hyp.7) + (1)
- $\mu_f \restriction^{\Sigma_f,\sigma} = \hat{\mu} \restriction^{\Sigma_1,\sigma}$ and $(\mu_f, \hat{r}) \restriction^{\bar{\Gamma},\sigma} = (\hat{\mu}, \hat{r}) \restriction^{\bar{\Gamma},\sigma}$ (26) - (1) + (20) + (23) + Confinement (Lemma 5.2)
- $\hat{\mu}' \restriction^{\Sigma_1',\sigma} = \mu_1' \restriction^{\Sigma_1',\sigma}$ and $(\hat{\mu}', r) \restriction^{\Gamma,\sigma} = (\mu_1', r) \restriction^{\Gamma,\sigma}$ (27) - (hyp.7) + (2)
- $\mu_f' \restriction^{\Sigma_f',\sigma} = \hat{\mu}' \restriction^{\Sigma_1',\sigma}$ and $(\mu_f', \hat{r}') \restriction^{\bar{\Gamma}',\sigma} = (\hat{\mu}', \hat{r}') \restriction^{\bar{\Gamma}',\sigma}$
  (28) - (2) + (20) + (24) + Confinement (Lemma 5.2)
- $\mu_1 \restriction^{\Sigma_1,\sigma} = \mu_1' \restriction^{\Sigma_1',\sigma}$ and $(\mu_1, r) \restriction^{\Gamma,\sigma} = (\mu_1', r) \restriction^{\Gamma,\sigma}$ (29) - (5)
- $\mu_f \restriction^{\Sigma_f,\sigma} = \mu_f' \restriction^{\Sigma_f',\sigma} \Leftrightarrow \mu_f, \Sigma_f \sim_\sigma \mu_f', \Sigma_f'$ **(30)** - (25)-(29)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu_f'$ **(31)** - (25)-(29)
- $lev(\dot{\tau}) \not\sqsubseteq \sigma$ (32) - (hyp.7) + (3)
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v_f'$ **(33)** - (32)

[Method Call] Suppose $e = e_0[e_1, P](e_2)$ for two expressions e_0 and e_1 (hyp.6). We conclude that there are eight memories μ_0, μ_1, μ_2, $\hat{\mu}$, μ_0', μ_1', μ_2', $\hat{\mu}'$, six type-based labellings Σ_0, Σ_1, Σ_2, Σ_0', Σ_1', and Σ_2', four references r_0, $\hat{r}$, r_0', and $\hat{r}'$, two strings m_1 and m_1', two values v_2 and v_2', two expressions $\hat{e}$ and $\hat{e}'$, three security types $\dot{\tau}_0$, $\dot{\tau}_1$, and $\dot{\tau}_2$, and security level σ', such that:

- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, r_0 \rangle$, $r \vdash \langle \mu_0, \Sigma_0, e_1 \rangle \Downarrow \langle \mu_1, \Sigma_1, m_1 \rangle$, $r \vdash \langle \mu_1, \Sigma_1, e_2 \rangle \Downarrow \langle \mu_2, \Sigma_2, v_2 \rangle$, $\hat{r} \vdash \langle \hat{\mu}, \Sigma_2, \hat{e} \rangle \Downarrow \langle \mu_f, \Sigma_f, v_f \rangle$, $r_m = \mathsf{Proto}(\mu_2, r_0, m_1)$, $r_f = \mu_2(r_m \cdot m_1)$, and $\langle \hat{\mu}, \hat{e}, \hat{r} \rangle = \mathsf{NewScope}(\mu_2, r_f, v_2, r_0, \Sigma_2)$ (1) - (hyp.2) + (hyp.6)
- $r \vdash \langle \mu', \Sigma', e_0 \rangle \Downarrow \langle \mu_0', \Sigma_0', r_0' \rangle$, $r \vdash \langle \mu_0', \Sigma_0', e_1 \rangle \Downarrow \langle \mu_1', \Sigma_1', m_1' \rangle$, $r \vdash \langle \mu_1', \Sigma_1', e_2 \rangle \Downarrow \langle \mu_2', \Sigma_2', v_2 \rangle$, $\hat{r}' \vdash \langle \hat{\mu}', \Sigma_2', \hat{e}' \rangle \Downarrow \langle \mu_f', \Sigma_f', v_f' \rangle$, $r_m' = \mathsf{Proto}(\mu_2', r_0', m_1')$, $r_f' = \mu_2'(r_m' \cdot m_1')$, and $\langle \hat{\mu}', \hat{e}', \hat{r}' \rangle = \mathsf{NewScope}(\mu_2', r_f', v_2', r_0', \Sigma_2')$ (2) - (hyp.3) + (hyp.6)
- $\Gamma, \sigma_{pc} \vdash e_i : \dot{\tau}_i, \sigma_i$: $i \in \{0, 1, 2\}$, $\pi_{\mathtt{type}}(\lceil\!\uparrow (\dot{\tau}_0, P)) = \langle \dot{\tau}_0'.\dot{\tau}_1' \xrightarrow{\hat{\sigma}} \dot{\tau}_2' \rangle^{\hat{\sigma}'}$, $\sigma' = \sigma_{pc} \sqcup \hat{\sigma}' \sqcup lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1)$, $\dot{\tau}_0^{\sigma'} \preceq \dot{\tau}_0'$, $\dot{\tau}_2^{\sigma'} \preceq \dot{\tau}_1'$, $\sigma' \sqsubseteq \hat{\sigma}$, and $\dot{\tau} = (\dot{\tau}_2')^{\sigma'}$ (3) - (hyp.1) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu_0', \Sigma_0', \Gamma, r \Vdash \mu_0 \sim_\sigma \mu_0'$, $lev(\dot{\tau}_0) \sqsubseteq \sigma \Rightarrow r_0 = r_0'$ (4) - (hyp.4) + (hyp.5) + (1) + (2) + (3) + **ih**
- $\mu_1, \Sigma_1 \sim_\sigma \mu_1', \Sigma_1', \Gamma, r \Vdash \mu_1 \sim_\sigma \mu_1'$, $lev(\dot{\tau}_1) \sqsubseteq \sigma \Rightarrow m_1 = m_1'$ (5) - (1) + (2) + (3) + (4) + **ih**
- $\mu_2, \Sigma_2 \sim_\sigma \mu_2', \Sigma_2', \Gamma, r \Vdash \mu_2 \sim_\sigma \mu_2'$, $lev(\dot{\tau}_2) \sqsubseteq \sigma \Rightarrow v_2 = v_2'$ (6) - (1) + (2) + (3) + (5) + **ih**

We consider two cases: either $\sigma' \sqsubseteq \sigma$ or $\sigma' \not\sqsubseteq \sigma$. Suppose $\sigma' \sqsubseteq \sigma$ (hyp.7). It follows that:

- $\sigma_{pc} \sqcup \hat{\sigma}' \sqcup lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqsubseteq \sigma$ (7) - (hyp.7) + (3)
- $r_0 = r_0'$ and $m_1 = m_1'$ (8) - (4) + (5) + (7)
- $m_1 = m_1' \in P$ (9) - (1) + (2) + (8) + Correct Annotation
- $\pi_{\mathtt{type}}(\lceil\!\uparrow (\dot{\tau}_0, m_1)) \preceq \pi_{\mathtt{type}}(\lceil\!\uparrow (\dot{\tau}_0, P)) = \langle \dot{\tau}_0'.\dot{\tau}_1' \xrightarrow{\hat{\sigma}} \dot{\tau}_2' \rangle^{\hat{\sigma}'}$ (10) - (3) + (9)
- $\Sigma_2(r_0) \curlyvee \Sigma_2'(r_0') \preceq \dot{\tau}_0$ (11) - (1) + (2) + (3) + Well-Labelling Preservation (Lemma 5.1)
- $\Sigma_2(r_0) = \Sigma_2'(r_0) \preceq \dot{\tau}_0$ (12) - (hyp.7) + (6)-(8) + (11)
- $lev(\Sigma_2(r_0)) = lev(\Sigma_2'(r_0)) \sqsubseteq lev(\dot{\tau}_0)$ (13) - (12)
- $\lfloor \Sigma_2(r_0) \rfloor = \lfloor \Sigma_2'(r_0) \rfloor = \lfloor \dot{\tau}_0 \rfloor$ (14) - (12)
- $\lceil\!\uparrow (\dot{\tau}_0, m_1) = \lceil\!\uparrow (\Sigma_2(r_0), m_1) = \lceil\!\uparrow (\Sigma_2'(r_0), m_1)$ (15) - (14)
- $\pi_{\mathtt{type}}(\lceil\!\uparrow (\Sigma_2(r_0), m_1)) = \pi_{\mathtt{type}}(\lceil\!\uparrow (\Sigma_2'(r_0), m_1)) = \pi_{\mathtt{type}}(\lceil\!\uparrow (\dot{\tau}_0, m_1))$ (16) - (15)
- $\pi_{\mathtt{type}}(\lceil\!\uparrow (\Sigma_2(r_0), m_1)) \sqsubseteq lev(\pi_{\mathtt{type}}(\lceil\!\uparrow (\Sigma_2(r_0), m_1)))$ (17) - Syntax of Types
- $lev(\pi_{\mathtt{type}}(\lceil\!\uparrow (\Sigma_2(r_0), m_1))) = lev(\pi_{\mathtt{type}}(\lceil\!\uparrow (\dot{\tau}_0, m_1))) \sqsubseteq \hat{\sigma}' \sqsubseteq \sigma$ (18) - (7) + (10) + (15)
- $\pi_{\mathtt{lev}}(\lceil\!\uparrow (\Sigma_2(r_0), m_1)) \sqsubseteq \sigma$ (19) - (17) + (18)
- $lev(\Sigma_2(r_0)) \sqsubseteq \sigma$ (20) - (7) + (12)
- $\pi_{\mathtt{lev}}(\lceil\!\uparrow (\Sigma_2(r_0), m_1)) \sqcup lev(\Sigma_2(r_0)) \sqsubseteq \sigma$ (21) - (19) + (20)
- $r_m = r_m'$ and $r_m \neq \mathsf{null} \Rightarrow lev(\Sigma_2(r_m)) = lev(\Sigma_2'(r_m')) \sqsubseteq \sigma$ (22) - (1) + (2) + (6) + (21) + Prototype-Chain Indistinguishability (Lemma B.2)
- $lev(\Sigma_2(r_m)) = lev(\Sigma_2'(r_m')) \sqsubseteq \sigma$ (23) - (1) + (2) + (22)
- $\Sigma_2(r_m) = \Sigma_2'(r_m)$ (24) - (6) + (22) + (23)
- $\lceil\!\uparrow (\Sigma_2(r_0), m_1) = \lceil\!\uparrow (\Sigma_2(r_m), m_1)$ (25) - (1) + Well-Labelling Preservation (Lemma 5.1)
- $\lceil\!\uparrow (\Sigma_2'(r_0), m_1) = \lceil\!\uparrow (\Sigma_2'(r_m), m_1)$ (26) - (2) + Well-Labelling Preservation (Lemma 5.1)
- $lev(\pi_{\mathtt{type}}(\lceil\!\uparrow (\Sigma_2(r_m), m_1))) = lev(\pi_{\mathtt{type}}(\lceil\!\uparrow (\Sigma_2'(r_m), m_1))) \sqsubseteq \sigma$ (27) - (18) + (25) + (26)
- $r_f = r_f'$ (28) - (1) + (2) + (6) + (8) + (22) + (23) + (27)
- $\Sigma_2(r_f) \curlyvee \Sigma_2'(r_f) \preceq \pi_{\mathtt{type}}(\lceil\!\uparrow (\dot{\tau}_0, P))$ (29) - (1) - (3) + Well-Labelling Preservation (Lemma 5.1)

- $\Sigma_2(r_f) = \Sigma_2'(r_f) \preceq \pi_{\texttt{type}}(\Gamma\!\!\uparrow\ (\dot\tau_0, P))$ (30) - (6) + (27) + (29)
- $\lfloor \Sigma_2(r_f) \rfloor \equiv \lfloor \Sigma_2'(r_f) \rfloor \equiv \lfloor \pi_{\texttt{type}}(\Gamma\!\!\uparrow\ (\dot\tau_0, P)) \rfloor \equiv \langle \dot\tau_0'.\dot\tau_1' \xrightarrow{\hat\sigma'} \dot\tau_2' \rangle$ (31) - (30)
- $lev(\Sigma_2(r_f)) = lev(\Sigma_2'(r_f)) \sqsubseteq \hat\sigma' \sqsubseteq \sigma$ (32) - (8) + (30)
- $\begin{cases} \mu_2(r_f \cdot \texttt{"@code"}) = \mu_2'(r_f \cdot \texttt{"@code"}) = \lambda^{\hat\Gamma, \Sigma_2(r_f)} x.\, \{\mathsf{var}^{\dot\tau_{y_1}, \cdots, \dot\tau_{y_n}}\ y_1, \cdots, y_n;\ \hat e\} \\ \mu_2(r_f \cdot \texttt{"@fscope"}) = \mu_2'(r_f \cdot \texttt{"@fscope"}) = \hat r = \hat r' \\ \hat\Gamma, \hat r \Vdash \mu_2 \sim_\sigma \mu_2' \\ \hat e = \hat e' \end{cases}$
  for some typing environment $\hat\Gamma$ and variables $x, y_1, \cdots, y_n$ (33) - (6)
- $\bar\Gamma, \hat\sigma \vdash \hat e : \dot\tau_2'$, where $\bar\Gamma = \hat\Gamma\,[\mathsf{this} \mapsto \dot\tau_0', x \mapsto \dot\tau_1', y_1 \mapsto \dot\tau_{y_1}, \cdots, y_n \mapsto \dot\tau_{y_n}]$
  (34) - (33) + Well-Labelling Preservation (Lemma 5.1)
- $lev(\dot\tau_0') \sqsubseteq \sigma \Rightarrow r_0 = r_0$ (35) - tautology
- $lev(\dot\tau_1') \sqsubseteq \sigma \Rightarrow lev(\dot\tau_2) \sqsubseteq \sigma$ (36) - (3)
- $lev(\dot\tau_1') \sqsubseteq \sigma \Rightarrow v_2 = v_2'$ (37) - (6) + (36)
- $\bar\Gamma, \hat r \Vdash \hat\mu \sim_\sigma \hat\mu'$ (38) - (1) + (2) + (6) + (32) + (35) + (37)
- $\hat\mu, \Sigma_2 \sim_\sigma \hat\mu', \Sigma_2'$ (39) - (1) + (2) + (6)
- $\mu_f, \Sigma_f \sim_\sigma \mu_f', \Sigma_f', \Gamma, r \Vdash \mu_f \sim_\sigma \mu_f', lev(\dot\tau_2') \sqsubseteq \sigma \Rightarrow v_f = v_f'$
  (40.1) - (1) + (2) + (34) + (39) + **ih**
- $lev(\dot\tau) \sqsubseteq \sigma \Rightarrow lev(\dot\tau_2') \sqsubseteq \sigma$ (40.2) - (3)
- $lev(\dot\tau) \sqsubseteq \sigma \Rightarrow v_f = v_f'$ **(40.3)** - (40.1) + (40.2)

Suppose $\sigma' \sqsubseteq \sigma$ (hyp.7). It follows that:

- $\hat\sigma \not\sqsubseteq \sigma$ (41) - (hyp.7) + (3)
- $\Sigma_2(r_f)\ \Upsilon\ \Sigma_2'(r_f') \preceq \pi_{\texttt{type}}(\Gamma\!\!\uparrow\ (\dot\tau_0, P))$ (42) - (1) - (3) + Well-Labelling Preservation (Lemma 5.1)
- $\lfloor \Sigma_2(r_f) \rfloor \equiv \lfloor \Sigma_2'(r_f') \rfloor \equiv \lfloor \pi_{\texttt{type}}(\Gamma\!\!\uparrow\ (\dot\tau_0, P)) \rfloor \equiv \langle \dot\tau_0'.\dot\tau_1' \xrightarrow{\hat\sigma'} \dot\tau_2' \rangle$ (43) - (42)
- $\begin{cases} \mu_2(r_f \cdot \texttt{"@code"}) = \lambda^{\hat\Gamma, \Sigma_2(r_f)} x.\, \{\mathsf{var}^{\dot\tau_{y_1}, \cdots, \dot\tau_{y_n}}\ y_1, \cdots, y_n;\ \hat e\} \\ \mu_2(r_f \cdot \texttt{"@fscope"}) = \hat r \\ \bar\Gamma \vdash \hat e : \dot\tau_2', \hat\sigma' \\ \bar\Gamma = \hat\Gamma\,[\mathsf{this} \mapsto \dot\tau_0', x \mapsto \dot\tau_1', y_1 \mapsto \dot\tau_{y_1}, \cdots, y_n \mapsto \dot\tau_{y_n}] \end{cases}$
  for some typing environment $\hat\Gamma$ and variables $x, y_1, \cdots, y_n$
  (44) - (1) + (3) + (43) + Well-Labelling Preservation (Lemma 5.1)
- $\begin{cases} \mu_2'(r_f' \cdot \texttt{"@code"}) = \lambda^{\hat\Gamma', \Sigma_2'(r_f')} x'.\, \left\{\mathsf{var}^{\dot\tau'_{y_1'}, \cdots, \dot\tau'_{y_k'}}\ y_1', \cdots, y_k';\ \hat e'\right\} \\ \mu_2'(r_f' \cdot \texttt{"@fscope"}) = \hat r' \\ \bar\Gamma' \vdash \hat e' : \dot\tau_2', \hat\sigma' \\ \bar\Gamma' = \hat\Gamma' \left[\mathsf{this} \mapsto \dot\tau_0', x' \mapsto \dot\tau_1', y_1' \mapsto \dot\tau'_{y_1'}, \cdots, y_n' \mapsto \dot\tau'_{y_n'}\right] \end{cases}$
  (45) - (2) + (3) + (43) + Well-Labelling Preservation (Lemma 5.1)
- $\hat\mu\ {\restriction}^{\Sigma_2,\sigma} = \mu_2\ {\restriction}^{\Sigma_2,\sigma}$ and $(\hat\mu, r)\ {\restriction}^{\Gamma,\sigma} = (\mu_2, r)\ {\restriction}^{\Gamma,\sigma}$ (46) - (1)
- $\mu_f\ {\restriction}^{\Sigma_f,\sigma} = \hat\mu\ {\restriction}^{\Sigma_2,\sigma}$ and $(\mu_f, \hat r)\ {\restriction}^{\bar\Gamma,\sigma} = (\hat\mu, \hat r)\ {\restriction}^{\bar\Gamma,\sigma}$ (47) - (1) + (44) + Confinement (Lemma 5.2)
- $\hat\mu'\ {\restriction}^{\Sigma_2',\sigma} = \mu_2'\ {\restriction}^{\Sigma_2',\sigma}$ and $(\hat\mu', r)\ {\restriction}^{\Gamma,\sigma} = (\mu_2', r)\ {\restriction}^{\Gamma,\sigma}$ (48) - (2)
- $\mu_f'\ {\restriction}^{\Sigma_f',\sigma} = \hat\mu'\ {\restriction}^{\Sigma_2',\sigma}$ and $(\mu_f', \hat r')\ {\restriction}^{\bar\Gamma',\sigma} = (\hat\mu', \hat r')\ {\restriction}^{\bar\Gamma',\sigma}$ (49) - (2) + (45) + Confinement (Lemma 5.2)
- $\mu_2\ {\restriction}^{\Sigma_2,\sigma} = \mu_2'\ {\restriction}^{\Sigma_2',\sigma}$ and $(\mu_2, r)\ {\restriction}^{\Gamma,\sigma} = (\mu_2', r)\ {\restriction}^{\Gamma,\sigma}$ (50) - (6)
- $\mu_f\ {\restriction}^{\Sigma_f,\sigma} = \mu_f'\ {\restriction}^{\Sigma_f',\sigma} \Leftrightarrow \mu_f, \Sigma_f \sim_\sigma \mu_f', \Sigma_f'$ **(51)** - (46)-(50)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu_f'$ **(52)** - (46)-(50)

- $lev(\dot{\tau}) \not\sqsubseteq \sigma$ (53) - (hyp.7) + (3)
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(54)** - (53)

[Property Deletion] Suppose $e = \mathsf{delete}\ e_0[p]$ for some expression e_0 and property p (hyp.6). It follows that there are two memories μ_0 and μ'_0, two references r_0 and r'_0, a security type $\dot{\tau}_0$, and a security level σ_0 such that:

- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_f, r_0 \rangle$, $\mu_f = \mu_0 \left[r_0 \mapsto \mu_0(r_0)|_{dom(\mu_0(r_0)-p)} \right]$, and $v_f = \mathsf{true}$ (1) - (hyp.2) + (hyp.6)
- $r \vdash \langle \mu', \Sigma', e_0 \rangle \Downarrow \langle \mu'_0, \Sigma'_f, r'_0 \rangle$, $\mu'_f = \mu'_0 \left[r'_0 \mapsto \mu'_0(r'_0)|_{dom(\mu'_0(r'_0)-p)} \right]$, and $v'_f = \mathsf{true}$ (2) - (hyp.3) + (hyp.6)
- $\Gamma, \sigma_{pc} \vdash e_0 : \dot{\tau}_0$, $\pi_{\mathtt{lev}}(\Pi\ (\dot{\tau}_0, p)) = \sigma_0$, $lev(\dot{\tau}_0) \sqsubseteq \sigma_0$, and $\dot{\tau} = \mathsf{PRIM}^{\perp}$ (3) - (hyp.1) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0, \Gamma, r \Vdash \mu_0 \sim_\sigma \mu'_0$, $lev(\dot{\tau}_0) \sqsubseteq \sigma \Rightarrow r_0 = r'_0$ (4) - (hyp.4) + (hyp.5) + (1) + (2) + (3) + **ih**
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(5)** - (1) + (2) + (4)
- $v_f = v'_f = \mathsf{true}$ (6) - (1) + (2)
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(7)** - (6)

We consider two cases: either $lev(\dot{\tau}_0) \sqsubseteq \sigma$ or $lev(\dot{\tau}_0) \not\sqsubseteq \sigma$. Suppose $lev(\dot{\tau}_0) \sqsubseteq \sigma$ (hyp.7). It follows that:

- $r_0 = r'_0$ (8) - (hyp.7) + (4)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(9)** - (1) + (2) + (4) + (8)

Suppose $lev(\dot{\tau}_0) \not\sqsubseteq \sigma$ (hyp.7). It follows that:

- $\sigma_0 = \pi_{\mathtt{lev}}(\Pi\ (\dot{\tau}_0, p)) \not\sqsubseteq \sigma$ (10) - (hyp.7) + (4)
- $\Sigma_f(r_0) \curlyvee \Sigma'_f(r'_0) \preceq \dot{\tau}_0$ (11) - (1)-(3) + Well-Labelling Preservation (Lemma 5.1)
- $\lfloor \Sigma_f(r_0) \rfloor \equiv \lfloor \Sigma'_f(r'_0) \rfloor \equiv \lfloor \dot{\tau}_0 \rfloor$ (12) - (11)
- $\pi_{\mathtt{lev}}(\Pi\ (\dot{\tau}_0, p)) = \pi_{\mathtt{lev}}(\Pi\ (\Sigma_f(r_0), p)) = \pi_{\mathtt{lev}}(\Pi\ (\Sigma'_f(r'_0), p))$ (13) - (12)
- $\pi_{\mathtt{lev}}(\Pi\ (\Sigma_f(r_0), p)) = \pi_{\mathtt{lev}}(\Pi\ (\Sigma'_f(r'_0), p)) \not\sqsubseteq \sigma$ (14) - (10) + (13)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(15)** - (1) + (2) + (4) + (14)

[Sequence] Suppose $e = e_0, e_1$ for two expressions e_0 and e_1 (hyp.6). We conclude that there are two memories μ_0 and μ'_0, two type-based labellings Σ_0 and Σ'_0, two values v_0 and v'_0, and two security types $\dot{\tau}_0$ and $\dot{\tau}_1$, such that:

- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, v_0 \rangle$ and $r \vdash \langle \mu_0, \Sigma_0, e_1 \rangle \Downarrow \langle \mu_f, \Sigma_f, v_f \rangle$ (1) - (hyp.2) + (hyp.6)
- $r \vdash \langle \mu', \Sigma', e_0 \rangle \Downarrow \langle \mu'_0, \Sigma'_0, v'_0 \rangle$ and $r \vdash \langle \mu'_0, \Sigma'_0, e'_1 \rangle \Downarrow \langle \mu'_f, \Sigma'_f, v'_f \rangle$ (2) - (hyp.3) + (hyp.6)
- $\Gamma, \sigma_{pc} \vdash e_0 : \dot{\tau}_0$, $\Gamma, \sigma_{pc} \vdash e_1 : \dot{\tau}_1$, and $\dot{\tau} = \dot{\tau}_1$ (3) - (hyp.1) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0, \Gamma, r \Vdash \mu_0 \sim_\sigma \mu'_0$, $lev(\dot{\tau}_0) \sqsubseteq \sigma \Rightarrow v_0 = v'_0$ (4) - (hyp.4) + (hyp.5) + (1) + (2) + (3) + **ih**
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f, \Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$, $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(5)** - (1) + (2) + (3) + (4) + **ih**

[Conditional Expression] Suppose $e = e_0\ ?\ (e_1) : (e_2)$ for three expressions e_0, e_1, and e_2 (hyp.6). We conclude that there are two memories μ_0 and μ'_0, two type-based labellings Σ_0 and Σ'_0, two values v_0 and v'_0, and two three types $\dot{\tau}_0$, $\dot{\tau}_1$, and $\dot{\tau}_2$, such that:

- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, v_0 \rangle$ and $r \vdash \langle \mu_0, \Sigma_0, e_i \rangle \Downarrow \langle \mu_f, \Sigma_f, v_f \rangle$ where: $v_0 \notin V_F \Rightarrow i = 1$ and $v_0 \in V_F \Rightarrow i = 2$ (1) - (hyp.2) + (hyp.6)

- $r \vdash \langle \mu', \Sigma', e_0 \rangle \Downarrow \langle \mu'_0, \Sigma'_0, v'_0 \rangle$ and $r \vdash \langle \mu'_0, \Sigma'_0, e_j \rangle \Downarrow \langle \mu'_f, \Sigma'_f, v'_f \rangle$ where: $v'_0 \notin V_F \Rightarrow j = 1$ and $v'_0 \in V_F \Rightarrow j = 2$ (2) - (hyp.3) + (hyp.6)
- $\Gamma, \sigma_{pc} \vdash e_0 : \dot{\tau}_0$, $\Gamma, \sigma_{pc} \sqcup lev(\dot{\tau}_0) \vdash e_i : \dot{\tau}_i$ for $i = 1, 2$, and $\dot{\tau} = (\dot{\tau}_1 \curlyvee \dot{\tau}_2)^{lev(\dot{\tau}_0)}$ (3) - (hyp.1) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0, \Gamma, r \Vdash \mu_0 \sim_\sigma \mu'_0$, $lev(\dot{\tau}_0) \sqsubseteq \sigma \Rightarrow v_0 = v'_0$ (4) - (hyp.4) + (hyp.5) + (1) + (2) + (3) + **ih**

We consider two cases: $lev(\dot{\tau}_0) \sqsubseteq \sigma$ and $lev(\dot{\tau}_0) \not\sqsubseteq \sigma$. Suppose $lev(\dot{\tau}_0) \sqsubseteq \sigma$ (hyp.7). It follows that:

- $v_0 = v'_0$ (5) - (hyp.7) + (4)
- $i = j$ (6) - (1) + (2) + (5)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f, \Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$, $lev(\dot{\tau}_i) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(7)** - (1) + (2) + (3) + (4) + (6) + **ih**
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow lev(\dot{\tau}_i) \sqsubseteq \sigma$ (8) - (3)
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(9)** - (7) + (8)

Suppose $lev(\dot{\tau}_0) \not\sqsubseteq \sigma$ (hyp.7)

- $\sigma_{pc} \sqcup lev(\dot{\tau}_0) \not\sqsubseteq \sigma$ (10) - (hyp.7) + (3)
- $\mu_f \restriction^{\Sigma_f, \sigma} = \mu_0 \restriction^{\Sigma_0, \sigma}$ and $(\mu_f, r) \restriction^{\Gamma, \sigma} = (\mu_0, r) \restriction^{\Gamma, \sigma}$ (11) - (1) + (10) + Confinement (Lemma 5.2)
- $\mu'_f \restriction^{\Sigma'_f, \sigma} = \mu'_0 \restriction^{\Sigma'_0, \sigma}$ and $(\mu'_f, r) \restriction^{\Gamma, \sigma} = (\mu'_0, r) \restriction^{\Gamma, \sigma}$ (12) - (2) + (10) + Confinement (Lemma 5.2)
- $\mu_0 \restriction^{\Sigma_0, \sigma} = \mu'_0 \restriction^{\Sigma'_0, \sigma}$ and $(\mu_0, r) \restriction^{\Gamma, \sigma} = (\mu'_0, r) \restriction^{\Gamma, \sigma}$ (13) - (4)
- $\mu_f \restriction^{\Sigma_f, \sigma} = \mu'_f \restriction^{\Sigma'_f, \sigma} \Leftrightarrow \mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(14)** - (11)-(13)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(15)** - (11)-(13)
- $lev(\dot{\tau}) \not\sqsubseteq \sigma$ (16) - (hyp.7)
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(17)** - (16)

[Function Literal] Suppose $e = \mathsf{function}^{\Gamma, \dot{\tau}, i}(x)\{\mathsf{var}^{\dot{\tau}_1, \cdots, \dot{\tau}_n}\ y_1, \cdots, y_n;\ \hat{e}\}$ (hyp.6).
Let $f = \lambda^{\Gamma, \dot{\tau}} x.\, \{\mathsf{var}^{\dot{\tau}_1, \cdots, \dot{\tau}_n}\ y_1, \cdots, y_n;\ \hat{e}\}$, we conclude that there are two references $\hat{r}$ and $\hat{r}'$, such that:

- $\mu_f = \mu\,[\hat{r} \mapsto [\texttt{"@fscope"} \mapsto r, \texttt{"@code"} \mapsto f]]$, $\Sigma_f = \Sigma\,[\hat{r} \mapsto \dot{\tau}]$, and $\hat{r} = \mathsf{fresh}(lev(\dot{\tau}))$ (1) - (hyp.1) + (hyp.2) + (hyp.6)
- $\mu'_f = \mu'\,[\hat{r}' \mapsto [\texttt{"@fscope"} \mapsto r, \texttt{"@code"} \mapsto f]]$, $\Sigma'_f = \Sigma'\,[\hat{r}' \mapsto \dot{\tau}]$, and $\hat{r}' = \mathsf{fresh}(lev(\dot{\tau}))$ (2) - (hyp.1) + (hyp.3) + (hyp.6)

We consider two cases: either $lev(\dot{\tau}) \sqsubseteq \sigma$ or $lev(\dot{\tau}) \not\sqsubseteq \sigma$. Suppose $lev(\dot{\tau}) \sqsubseteq \sigma$ (hyp.7):

- $\hat{r} = \hat{r}'$ (3) - (hyp.4) + (hyp.7) + (1) + (2)
- $\mu_f \restriction^{\Sigma_f, \sigma} = \mu \restriction^{\Sigma, \sigma} \cup \{(\hat{r}, f, r, (\mu, r) \restriction^{\Gamma, \sigma})\}$ (4) - (hyp.7) + (1)
- $\mu'_f \restriction^{\Sigma'_f, \sigma} = \mu' \restriction^{\Sigma', \sigma} \cup \{(\hat{r}, f, r, (\mu', r) \restriction^{\Gamma, \sigma})\}$ (5) - (hyp.7) + (1)
- $\mu \restriction^{\Sigma, \sigma} = \mu' \restriction^{\Sigma', \sigma}$ (6) - (hyp.4)
- $(\mu, r) \restriction^{\Gamma, \sigma} = (\mu', r) \restriction^{\Gamma, \sigma}$ (7) - (hyp.5)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(8)** - (4)-(7)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(9)** - (1) + (2)
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(10)** - (1) + (2) + (3)

Suppose $lev(\dot{\tau}) \not\sqsubseteq \sigma$ (hyp.7):

- $\mu_f \restriction^{\Sigma_f,\sigma} = \mu \restriction^{\Sigma,\sigma}$ (11) - (hyp.7) + (1)
- $\mu'_f \restriction^{\Sigma'_f,\sigma} = \mu' \restriction^{\Sigma',\sigma}$ (12) - (hyp.7) + (2)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(13)** - (hyp.4) + (11) + (12)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(14)** - (1) + (2)
- $lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ **(15)** - (hyp.7)

□

B.2 Soundness of the Hybrid Type System

The following lemma states that the execution of a typable expression preserves the well-typing predicate for memories. In other words, the execution of a typable expression in a well-typed memory always generates a well-typed memory.

Lemma B.7 (Well-Typing Preservation)**.** *For any two memories μ and μ', type-based labellings Σ and Σ', reference r, expressions e, e', and e'', value v, typing environment Γ, level set L, and type set T, such that:*

- $\Gamma, L \vdash e \rightsquigarrow {}^{e'}/_{e''} : T$ *(hyp.1)*
- $r \vdash \langle \mu, \Sigma, e\rangle \Downarrow \langle \mu', \Sigma', v\rangle$ *(hyp.2)*
- μ *is well-typed by* Σ *(hyp.3)*

It holds that: μ' is well-typed by Σ' and if $v \in$ `Ref`, it holds that: $\forall_{(\dot{\tau},\omega)\in T}\ \mu', r \vDash \omega \Rightarrow \Sigma'(v) \preceq \dot{\tau}$.

Proof: The result follows by induction on (hyp.2). □

Suppose that a program is typable using the hybrid type system with a level set L (that represents the possible levels of the program counter). Lemma B.8 states that for every pair $(\sigma, \omega) \in L$, if the assertion ω holds in the final memory, then the execution of the instrumented expression generated by the type system is confined at level σ. In other words, if σ is a *possible context level*, whenever its corresponding assertion holds, the execution of the instrumented expression generated by the type system only changes the resources whose levels are higher than or equal to σ.

Lemma B.8 (Confinement)**.** *For any two memories μ and μ', type-based labellings Σ and Σ', reference r, expressions e, e', and e'', value v, typing environment Γ, level set L, type set T, and security level σ, such that:*

- $\Gamma, L \vdash e \rightsquigarrow {}^{e'}/_{e''} : T$ *(hyp.1)*
- $r \vdash \langle \mu, \Sigma, e\rangle \Downarrow \langle \mu', \Sigma', v\rangle$ *(hyp.2)*
- μ *is well-typed by* Σ *(hyp.3)*

It holds that: $\forall_{(\sigma',\omega)\in L}\ \mu', r \vDash \omega\ \wedge\ \sigma' \not\sqsubseteq \sigma \Rightarrow \mu \restriction^{\Sigma,\sigma} = \mu' \restriction^{\Sigma',\sigma}\ \wedge (\mu, r) \restriction^{\Gamma,\sigma} = (\mu', r) \restriction^{\Gamma,\sigma}$.

Proof: The result follows by induction on (hyp.2). □

Theorem 5.2 - Transparency

Proof: We have to prove that given that:

- $\Gamma \vdash e \rightsquigarrow e'/e'' : T, L$ (hyp.1)

- $r \vdash \langle \mu', \Sigma, e' \rangle \Downarrow \langle \mu'_f, \Sigma_f, v_f \rangle$ (hyp.2)
- $\mu \simeq_{hts} \mu'$ (hyp.3)

then, it holds that there exists a memory μ_f such that:

- $r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu_f, \Sigma_f, v_f \rangle$;
- $\mu_f \simeq_{hts} \mu'_f$;
- $\big(e'' \in \texttt{Prim} \wedge e'' = v_f\big) \vee \big(e'' \in \texttt{Var} \wedge m_{e''} = \mathsf{string}(e'') \wedge \mu'_f(r \cdot e'') = v_f\big)$

We proceed by induction on the derivation of (*hyp.*2). For simplicity, we structure our analysis of the cases according to the last rule used in the typing of e.

[VAL] $e = v$ for some value v (hyp.4). We conclude that:

- $e' = v$ and $e'' = v$ (1) - (hyp.1) + (hyp.4)
- $v_f = v$ (2) - (hyp.2) + (hyp.4)
- $\mu'_f = \mu'$ and $\Sigma_f = \Sigma$ (3) - (hyp.2) + (hyp.4)

If we make $\mu_f = \mu$ (hyp.5), we conclude that:

- $r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu_f, \Sigma_f, v_f \rangle$ **(4)** - (hyp.4) + (hyp.5) + (2) + (3)
- $\mu_f \simeq_{hts} \mu'_f$ **(5)** - (hyp.3) + (hyp.5) + (3)
- $e'' \in \texttt{Prim} \wedge e'' = v_f$ **(6)** - (1) + (2)

[THIS] $e = \mathsf{this}^i$ (hyp.4). We conclude that:

- $e' = \$v_i = \mathsf{this}$ and $e'' = \$v_i$ (1) - (hyp.1) + (hyp.4)
- $v_f = \mu'(r \cdot \texttt{"@this"})$ (2) - (hyp.2) + (hyp.4)
- $\mu'_f = \mu'[r \cdot \$v_i \mapsto v_f]$ and $\Sigma_f = \Sigma$ (3) - (hyp.2) + (hyp.4)
- $e'' \in \texttt{Var} \wedge m_i = \mathsf{string}(\$v_i) \wedge \mu'_f(r \cdot m_i) = v_f$ **(4)** - (1) + (2)
- $\mu'(r \cdot \texttt{"@this"}) = \mu(r \cdot \texttt{"@this"})$ (5) - (hyp.3)

If we make $\mu_f = \mu$ (hyp.5), we conclude that:

- $r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu_f, \Sigma_f, v_f \rangle$ **(6)** - (hyp.4) + (hyp.5) + (2) + (5)
- $\mu_f \simeq_{hts} \mu'_f$ **(7)** - (hyp.3) + (hyp.5) + (3)

[VARIABLE] $e = x^i$ for some variable x and index i (hyp.4). Letting $m_x = \mathsf{string}(x)$ and $m_i = \mathsf{string}(\$v_i)$, we conclude that there is a reference r_x such that:

- $e' = \$v_i = x$ and $e'' = \$v_i$ (1) - (hyp.1) + (hyp.4)
- $v_f = \mu'(r_x \cdot m_x)$ and $r_x = \mathsf{Scope}(\mu', r, x)$ (2) - (hyp.2) + (1)
- $\mu'_f = \mu'[r \cdot m_i \mapsto v_f]$ and $\Sigma_f = \Sigma$ (3) - (hyp.2) + (hyp.4)
- $e'' \in \texttt{Var} \wedge \mu'_f(r \cdot \$v_i) = v_f$ **(4)** - (1) + (2)
- $r_x = \mathsf{Scope}(\mu, r, x)$ and $\mu(r_x \cdot m_x) = \mu'(r_x \cdot m_x)$ (5) - (hyp.3) + (2)

Letting $\mu_f = \mu$ (hyp.5), we conclude that:

- $r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu_f, \Sigma_f, v_f \rangle$ **(6)** - (hyp.4) + (hyp.5) + (3) + (5)
- $\mu_f \simeq_{hts} \mu'_f$ **(7)** - (hyp.3) + (hyp.5) + (3)

[Binary Operation] $e = e_0\ \mathsf{op}^j\ e_1$ for two expressions e_0 and e_1 and index j (hyp.4). We conclude that there are four memories μ_0, μ_0', μ_1, and μ_1', a type-based labelling Σ_0, and two values v_0 and v_1 such that:

- $\Gamma \vdash e_i \rightsquigarrow e_i'/e_i'' : T_i, L_i$, where $i \in \{0,1\}$ and $e' = e_0', e_1', \$v_j = e_0''\ \mathsf{op}\ e_1''$ (1) - (hyp.1) + (hyp.4)
- $r \vdash \langle \mu', \Sigma, e_0' \rangle \Downarrow \langle \mu_0', \Sigma_0, v_0 \rangle$ and $r \vdash \langle \mu_0', \Sigma_0, e_1' \rangle \Downarrow \langle \mu_1', \Sigma_f, v_1 \rangle$ (2) - (hyp.2) + (hyp.4)
- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, v_0 \rangle$, $\mu_0 \simeq_{hts} \mu_0'$, and: $\big(e_0'' \in \mathtt{Prim} \wedge e_0'' = v_0\big) \vee \big(e_0'' \in \mathtt{Var} \wedge \mu_0'(r \cdot \mathsf{string}(e_0'')) = v_0\big)$ (3) - (hyp.3) + (1) + (2) + **ih**
- $r \vdash \langle \mu_0, \Sigma_0, e_1 \rangle \Downarrow \langle \mu_1, \Sigma_1, v_1 \rangle$, $\mu_1 \simeq_{hts} \mu_1'$, and: $\big(e_1'' \in \mathtt{Prim} \wedge e_1'' = v_1\big) \vee \big(e_1'' \in \mathtt{Var} \wedge \mu_1'(r \cdot \mathsf{string}(e_1'')) = v_1\big)$ (4) - (hyp.3) + (1) + (2) + **ih**
- $r \vdash \langle \mu_1', \Sigma_f, e_0'' \rangle \Downarrow \langle \mu_1'', \Sigma_f, v_0 \rangle$ and $r \vdash \langle \mu_1', \Sigma_f, e_1'' \rangle \Downarrow \langle \mu_1'', \Sigma_f, v_1 \rangle$ (5) - (3) + (4) + Invariance of Bookkeeping Expressions
- $v_f = v_0\ \mathsf{op}\ v_1$ and $\mu_f' = \mu_1'[r \cdot \mathsf{string}(\$v_j) \mapsto v_f]$ (6) - (1) + (2) + (5)
- $e'' \in \mathtt{Var} \ \wedge\ \mu_f'(r \cdot \mathsf{string}(\$v_j)) = v_f$ **(7)** - (1) + (6)
- $r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu_f, \Sigma_f, v_f \rangle$ **(8)** - (hyp.4) + (3) + (4)
- $\mu_f \simeq_{hts} \mu_f'$ **(9)** - (4) + (6)

[Object Literal] $e = \{\,\}^{\dot\tau, i}$ for an index i and a type $\dot\tau$ (hyp.4). We conclude that:

- $e' = \$v_i = \{\,\}^{\tau}$ and $e'' = \$v_i$ (1) - (hyp.1) + (hyp.4)
- $v_f = r_f = \mathsf{fresh}(lev(\dot\tau))$ (2) - (hyp.2) + (1)
- $\mu_f' = \mu[r_f \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}], r \mapsto \mu(r)[\mathsf{string}(\$v_i) \mapsto r_f]]$ and $\Sigma_f = \Sigma\,[r_f \mapsto \dot\tau]$ (3) - (hyp.2)
- $e'' \in \mathtt{Var} \ \wedge\ \mu_f'(r \cdot \mathsf{string}(\$v_i)) = v_f$ **(4)** - (1) - (3)

Letting $\mu_f = \mu(r \cdot \mathsf{string}(\$v_i))$ (hyp.5), we conclude that:

- $r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu_f, \Sigma_f, v_f \rangle$ **(5)** - (hyp.4) + (hyp.5) + (2) + (3)
- $\mu_f \simeq_{hts} \mu_f'$ **(6)** - (hyp.3) + (hyp.5) + (3)

[Variable Assignment] $e = x = e_0$ for some variable x and expression e_0 (hyp.4). Letting $m_x = \mathsf{string}(x)$, we conclude that there are two memories μ_0 and μ_0' such that:

- $e' = e_0', \mathsf{Wrap}(\omega, x = e_0'')$ and $e'' = e_0''$ and $\Gamma \vdash e_0 \rightsquigarrow e_0'/e_0'' : T_0, L_0$ (1) - (hyp.1) + (hyp.4)
- $r \vdash \langle \mu', \Sigma, e_0' \rangle \Downarrow \langle \mu_0', \Sigma_f, v_0 \rangle$ and $\mu_f' = \mu_0'[r \cdot m_x \mapsto v_0]$ (2) - (hyp.2) + (1)
- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, v_0 \rangle$, $\mu_0 \simeq_{hts} \mu_0'$, and: $\big(e_0'' \in \mathtt{Prim} \wedge e_0'' = v_0\big) \vee \big(e_0'' \in \mathtt{Var} \wedge \mu_0'(r \cdot \mathsf{string}(e_0'')) = v_0\big)$ (3) - (hyp.3) + (1) + (2) + **ih**
- $\mu_f' = \mu_0'[r \cdot m_x \mapsto v_0]$ (4) - (hyp.2) + (3)
- $e_0'' \in \mathtt{Prim} \wedge e_0'' = v_0 \ \vee\ e_0'' \in \mathtt{Var} \wedge \mu_f'(r \cdot e_0'') = v_0$ **(5)** - (3) + (4)

Letting $\mu_f = \mu_0[r \cdot m_x \mapsto v_0]$ (hyp.5), we conclude that:

- $r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu_f, \Sigma_f, v_f \rangle$ **(6)** - (hyp.4) + (hyp.5) + (3)-(5)
- $\mu_f \simeq_{hts} \mu_f'$ **(7)** - (hyp.3) + (hyp.5) + (3)-(5)

[Property Look-up] $e = e_0[e_1, P]^j$ for two expressions e_0 and e_1 (hyp.4). It follows that there are three memories μ_0, μ_1, μ_0', and μ_1', a labelling Σ_0, two references r_0 and $\hat r$, and a string m_1 such that:

- $\Gamma \vdash e_i \rightsquigarrow e_i'/e_i'' : T_i, L_i$ for $i = 0, 1$ and $e' = e_0', e_1', \$v_j = e_0''[e_1'']$ (1) - (hyp.1) + (hyp.4)
- $r \vdash \langle \mu', \Sigma, e_0' \rangle \Downarrow \langle \mu_0', \Sigma_0, r_0 \rangle$, and $r \vdash \langle \mu_0', \Sigma_0, e_1' \rangle \Downarrow \langle \mu_1', \Sigma_f, m_1 \rangle$ (2) - (hyp.2) + (hyp.4)
- $r \vdash \langle \mu, \Sigma, e_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, r_0 \rangle$, $\mu_0 \simeq_{hts} \mu_0'$, and:
  $\big(e_0'' \in \texttt{Prim} \wedge e_0'' = r_0\big) \vee \big(e_0'' \in \texttt{Var} \wedge \mu_0'(r \cdot \mathsf{string}(e_0'')) = r_0\big)$
  (3) - (hyp.3) + (1) + (2) + **ih**
- $r \vdash \langle \mu_0, \Sigma_0, e_1 \rangle \Downarrow \langle \mu_f, \Sigma_f, m_1 \rangle$, $\mu_f \simeq_{hts} \mu_1'$, and:
  $\big(e_1'' \in \texttt{Prim} \wedge e_1'' = m_1\big) \vee \big(e_1'' \in \texttt{Var} \wedge \mu_1'(r \cdot \mathsf{string}(e_1'')) = m_1\big)$ (4) - (1) - (3) + **ih**
- $\mu_f' = \mu_1'[r \cdot \mathsf{string}(\$v_j) \mapsto v_f]$, $\hat{r} = \mathsf{Proto}(\mu_1', r_0, m_1)$, and $v_f = \mu_1'(\hat{r} \cdot m_1)$ (5) - (hyp.2) + (2) - (4)
- $\big(\hat{r} = \mathsf{Proto}(\mu_f, r_0, m_1) \wedge \mu_1(\hat{r} \cdot m_1) = v_f\big) \vee \big(\mathsf{null} = \mathsf{Proto}(\mu_f, r_0, m_1) \wedge v_f = \mathsf{undefined}\big)$
  (6) - (4) + (5)
- $r \vdash \langle \mu, \Sigma, e \rangle \Downarrow \langle \mu_f, \Sigma_f, v_f \rangle$ **(7)** - (hyp.4) + (3) + (4) + (6)
- $\mu_f \simeq_{hts} \mu_f'$ **(8)** - (4) + (5)

The remaining cases are proven in a similar fashion.

Theorem 5.3 - Noninterference - Hybrid Type System

Proof: We have to prove that given that:

- $\Gamma, L_{pc} \vdash e \rightsquigarrow {}^{e'}/_{e''} : T$ (hyp.1)
- $r \vdash \langle \mu, \Sigma, e' \rangle \Downarrow \langle \mu_f, \Sigma_f, v_f \rangle$ (hyp.2)
- $r \vdash \langle \mu', \Sigma', e' \rangle \Downarrow \langle \mu_f', \Sigma_f', v_f' \rangle$ (hyp.3)
- $\mu, \Sigma \sim_\sigma \mu', \Sigma'$ (hyp.4)
- $\Gamma, r \Vdash \mu \sim_\sigma \mu'$ (hyp.5)

then, it holds that:

- $\mu_f, \Sigma_f \sim_\sigma \mu_f', \Sigma_f'$,
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu_f'$,
- for all $(\dot{\tau}, \omega) \in T$, if $lev(\dot{\tau}) \sqsubseteq \sigma$ then: $\mu_f, r \vDash \omega \Leftrightarrow \mu_f', r \vDash \omega$ and $\mu, r \vDash \omega \Rightarrow v_f = v_f'$.

We proceed by induction on the derivation of (*hyp*.2). For simplicity, we structure our analysis of the cases according to the last rule used in the typing of e.

[Val] Suppose $e = v$ for some value v (hyp.6). We conclude that:

- $e' = v$ (1) - (hyp.1) + (hyp.6)
- $v_f = v_f' = v$ (2) - (hyp.2) + (hyp.3) + (hyp.6)
- $\mu_f = \mu$, $\mu_f' = \mu'$, $\Sigma_f = \Sigma$, $\Sigma_f' = \Sigma'$ (3) - (hyp.2) + (hyp.3) + (hyp.6) + (1)
- $\mu_f, \Sigma_f \sim_\sigma \mu_f', \Sigma_f'$ **(4)** - (hyp.4) + (3)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu_f'$ **(5)** - (hyp.5) + (3)
- $T = \{(\mathsf{PRIM}^\perp, \mathsf{true})\}$ (6) - (hyp.1) + (hyp.6)
- $\mu_f, r \vDash \mathsf{true}$ and $\mu_f', r \vDash \mathsf{true}$ (7) - tautology
- $\mu_f, r \vDash \mathsf{true} \Rightarrow v_f = v_f'$ **(8)** - (2)
- $\mu_f, r \vDash \mathsf{true} \Leftrightarrow \mu_f', r \vDash \mathsf{true}$ **(9)** - (7)

[This] Suppose $e = \mathsf{this}$ (hyp.6). We conclude that:

- $e' = \$v_i = \mathsf{this}$ (1) - (hyp.1) + (hyp.6)
- $v_f = \mu(r \cdot \texttt{"@this"})$ and $v'_f = \mu'(r \cdot \texttt{"@this"})$ (2) - (hyp.2) + (hyp.3) + (1)
- $lev(\Gamma(\mathsf{this})) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ (3) - (hyp.5) + (2)
- $\mu_f = \mu$, $\mu'_f = \mu'$, $\Sigma_f = \Sigma$, and $\Sigma'_f = \Sigma'$. (4) - (hyp.2) + (hyp.3) + (1)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(5)** - (hyp.4) + (4)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(6)** - (hyp.5) + (4)
- $T = \{(\Gamma(\mathsf{this}), \mathsf{true})\}$ (7) - (hyp.1) + (hyp.6)

In order to prove the third claim of the lemma, suppose that $lev(\Gamma(\mathsf{this})) \sqsubseteq \sigma$ (hyp.7). It follows that:

- $\mu_f, r \vDash \mathsf{true} \Leftrightarrow \mu'_f, r \vDash \mathsf{true}$ **(8)** - tautology
- $v_f = v'_f$ (9) - (hyp.7) + (3)
- $\mu_f, r \vDash \mathsf{true} \Rightarrow v_f = v'_f$ **(10)** - (9)

[Variable] Suppose $e = x^i$, for some variable x and index i (hyp.6). Let $m_x = \mathsf{string}(x)$, we conclude that there are two references r_x and r'_x such that:

- $e' = \$v_i = x$ (1) - (hyp.2) + (hyp.6)
- $\mu = \mu_f$, $\Sigma = \Sigma_f$, $v_f = \mu(r_x \cdot m_x)$, and $r_x = \mathsf{Scope}(\mu, r, x)$ (2) - (hyp.2) + (1)
- $\mu' = \mu'_f$, $\Sigma' = \Sigma'_f$, $v'_f = \mu'(r'_x \cdot x)$, and $r'_x = \mathsf{Scope}(\mu', r, x)$ (3) - (hyp.3) + (1)
- $lev(\Gamma(x)) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ (4) - (hyp.5) + (2) + (3)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(5)** - (hyp.4) + (2) + (3)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(6)** - (hyp.5) + (2) + (3)
- $T = \{(\Gamma(x), \mathsf{true})\}$ (7) - (hyp.1) + (hyp.6)

Suppose that $lev(\Gamma(x)) \sqsubseteq \sigma$ (hyp.7). It follows that:

- $\mu_f, r \vDash \mathsf{true} \Leftrightarrow \mu'_f, r \vDash \mathsf{true}$ **(8)** - tautology
- $v_f = v'_f$ (9) - (hyp.7) + (4)
- $\mu_f, r \vDash \mathsf{true} \Rightarrow v_f = v'_f$ **(10)** - (9)

[Binary Operation] Suppose $e = e_0\ \mathsf{op}^j\ e_1$ for two exprs. e_0 and e_1 (hyp.6). We conclude that there are four memories μ_0, μ_1, μ'_0, and μ'_1, four type-based labellings Σ_0 and Σ'_0, four values v_0, v_1, v'_0, and v'_1, two type sets T_0 and T_1, four expressions e'_0, e''_0, e'_1, e''_1 such that:

- $\Gamma, L_{pc} \vdash e_i \rightsquigarrow {}^{e'_i}/_{e''_i} : T_i$ for $i \in \{0, 1\}$, $e' = e'_0, e'_1, \$v_j = e''_0\ \mathsf{op}\ e''_1$, and $T = T_0 \oplus_\Upsilon T_1$. (1) - (hyp.1) + (hyp.6)
- $r \vdash \langle \mu, \Sigma, e'_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, v_0 \rangle$, $r \vdash \langle \mu_0, \Sigma_0, e'_1 \rangle \Downarrow \langle \mu_1, \Sigma_f, v_1 \rangle$, and $v_f = v_0\ \mathsf{op}\ v_1$ (2) - (hyp.2) + (1)
- $r \vdash \langle \mu', \Sigma', e'_0 \rangle \Downarrow \langle \mu'_0, \Sigma'_0, v'_0 \rangle$, $r \vdash \langle \mu'_0, \Sigma'_0, e'_1 \rangle \Downarrow \langle \mu'_1, \Sigma'_f, v'_1 \rangle$, and $v'_f = v'_0\ \mathsf{op}\ v'_1$ (3) - (hyp.3) + (1)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0$, $\Gamma, r \Vdash \mu_0 \sim_\sigma \mu'_0$, and:

$$\forall_{(\dot{\tau}_0, \omega_0) \in T_0}\ lev(\dot{\tau}_0) \sqsubseteq \sigma \Rightarrow (\mu_0, r \vDash \omega_0 \Leftrightarrow \mu'_0, r \vDash \omega_0) \wedge (\mu_0, r \vDash \omega_0 \Rightarrow v_0 = v'_0)$$

(4) - (hyp.4) + (hyp.5) + (1) + (2) + (3) + **ih**

- $\mu_1, \Sigma_1 \sim_\sigma \mu_1', \Sigma_1', \Gamma, r \Vdash \mu_1 \sim_\sigma \mu_1'$, and:

$$\forall_{(\dot{\tau}_1,\omega_1)\in T_1}\ lev(\dot{\tau}_1) \sqsubseteq \sigma \Rightarrow (\mu_1, r \vDash \omega_1 \Leftrightarrow \mu_1', r \vDash \omega_1) \wedge (\mu_1, r \vDash \omega_1 \Rightarrow v_1 = v_1')$$

(5) - (1) + (2) + (3) + (4) + **ih**

- $\mu_f, \Sigma_f \sim_\sigma \mu_f', \Sigma_f'$ and $\Gamma, r \Vdash \mu_f \sim_\sigma \mu_f'$ **(6)** - (2) + (3) + (5)
- $\forall_{(\dot{\tau},\omega)\in T}\ \forall_{\hat{\mu},\hat{r}}\ \hat{\mu}, \hat{r} \vDash \omega \Leftrightarrow \exists_{(\dot{\tau}_0,\omega_0)\in T_0}\ \exists_{(\dot{\tau}_1,\omega_1)\in T_1} \hat{\mu}, \hat{r} \vDash (\omega_0 \wedge \omega_1) \wedge \dot{\tau} = \dot{\tau}_0 \curlyvee \dot{\tau}_1$ (7) - (1)

Suppose that $(\dot{\tau}, \omega) \in T$ (hyp.7), $lev(\dot{\tau}) \sqsubseteq \sigma$ (hyp.8), and $\mu_f, r \vDash \omega$ (hyp.9). It follows that there are $(\dot{\tau}_0, \omega_0) \in T_0$ and $(\dot{\tau}_1, \omega_1) \in T_1$ such that:

- $\mu_f, r \vDash (\omega_0 \wedge \omega_1)$ and $\dot{\tau} = \dot{\tau}_0 \curlyvee \dot{\tau}_1$ (8) - (hyp.7) + (hyp.8) + (7)
- $\mu_f, r \vDash \omega_0$, $\mu_f, r \vDash \omega_1$, and $\dot{\tau} = \dot{\tau}_0 \curlyvee \dot{\tau}_1$. (9) - (8)
- $\mu_f, r \vDash \omega_0 \Leftrightarrow \mu_0, r \vDash \omega_0$ and $\mu_f, r \vDash \omega_1 \Leftrightarrow \mu_1, r \vDash \omega_1$
(10) - (hyp.7) + (1) + (2) + Invariance of Dynamic Assertions
- $\mu_f', r \vDash \omega_0 \Leftrightarrow \mu_0', r \vDash \omega_0$ and $\mu_f', r \vDash \omega_1 \Leftrightarrow \mu_1', r \vDash \omega_1$
(11) - (hyp.7) + (1) + (3) + Invariance of Dynamic Assertions
- $\mu_0, r \vDash \omega_0$ and $\mu_1, r \vDash \omega_1$ (12) - (9) + (10)
- $lev(\dot{\tau}_0) \sqsubseteq \sigma$ and $lev(\dot{\tau}_1) \sqsubseteq \sigma$ (13) - (hyp.8) + (9)
- $\mu_0', r \vDash \omega_0$ and $v_0 = v_0'$ (14) - (4) + (12) + (13)
- $\mu_1', r \vDash \omega_1$ and $v_1 = v_1'$ (15) - (5) + (12) + (13)
- $\mu_f', r \vDash \omega_0$ and $\mu_f', r \vDash \omega_1$ (16) - (11) + (14) + (15)
- $\mu_f', r \vDash \omega_0 \wedge \omega_1$ **(17)** - (16)
- $v_f = v_f'$ **(18)** - (2) + (3) + (14) + (15)

[Object Literal] Suppose $e = \{\,\}^{\dot{\tau},i}$ for an index i and a type $\dot{\tau}$ (hyp.6). We conclude that there are two references $\hat{r}$ and $\hat{r}'$ such that:

- $T = \{(\dot{\tau}, \mathsf{true})\}$ and $e' = \$v_i = \{\,\}^\tau$ (1) - (hyp.1) + (hyp.6)
- $\hat{r} = \mathsf{fresh}(lev(\dot{\tau}))$, $\hat{\mu} = \mu\,[\hat{r} \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}]]$, $\Sigma_f = \Sigma\,[\hat{r} \mapsto \dot{\tau}]$, and $v_f = \hat{r}$
(2) - (hyp.2) + (hyp.6)
- $\hat{r}' = \mathsf{fresh}(lev(\dot{\tau}'))$, $\hat{\mu}' = \mu'\,[\hat{r}' \mapsto [\texttt{"\_prot\_"} \mapsto \mathsf{null}]]$, $\Sigma_f' = \Sigma'\,[\hat{r}' \mapsto \dot{\tau}]$, and $v_f' = \hat{r}'$
(3) - (hyp.3) + (hyp.6)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu_f'$ **(4)** - (hyp.5) + (2) + (3)

Suppose that $(\dot{\tau}', \omega) \in T$ (hyp.7), it follows that $\dot{\tau}' = \dot{\tau}$ and $\omega = \mathsf{true}$. We consider two cases: either $lev(\dot{\tau}) \sqsubseteq \sigma$ or $lev(\dot{\tau}) \not\sqsubseteq \sigma$. Suppose $lev(\dot{\tau}) \sqsubseteq \sigma$ (hyp.8):

- $\hat{r} = \hat{r}'$ (5) - (hyp.4) + (hyp.8) + (2) + (3)
- $\mu_f\,{\restriction}^{\Sigma_f,\sigma} = \mu\,{\restriction}^{\Sigma,\sigma} \cup \{(\hat{r}, \dot{\tau}), (\hat{r}, \texttt{"\_prot\_"}, \mathsf{null}), (\hat{r}, \texttt{"\_prot\_"})\}$ (6) - (hyp.8) + (2)
- $\mu_f'\,{\restriction}^{\Sigma_f',\sigma} = \mu'\,{\restriction}^{\Sigma',\sigma} \cup \{(\hat{r}, \dot{\tau}), (\hat{r}, \texttt{"\_prot\_"}, \mathsf{null}), (\hat{r}, \texttt{"\_prot\_"})\}$ (7) - (hyp.8) + (3) + (5)
- $\mu_f, \Sigma_f \sim_\sigma \mu_f', \Sigma_f'$ **(8)** - (hyp.4) + (6) + (7)
- $\mu_f, r \vDash \mathsf{true} \Leftrightarrow \mu_f', r \vDash \mathsf{true}$ **(9)** - tautology
- $v_f = v_f'$ (10) - (2) + (3) + (5)
- $\mu_f, r \vDash \mathsf{true} \Rightarrow v_f = v_f'$ **(11)** - (10)

Suppose $lev(\dot{\tau}) \not\sqsubseteq \sigma$ (hyp.8):

- $\mu_f\,{\restriction}^{\Sigma_f,\sigma} = \mu\,{\restriction}^{\Sigma,\sigma}$ (12) - (hyp.8) + (2)

- $\mu'_f \restriction^{\Sigma'_f,\sigma} = \mu' \restriction^{\Sigma',\sigma}$ (13) - (hyp.8) + (3)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(13)** - (hyp.4) + (12) + (13)

[Variable Assignment] Suppose $e = x = e_0$ for some variable e and expression e_0 (hyp.6). Let $m_x = \mathsf{string}(x)$, we conclude that there are two memories μ_0 and μ'_0, two type-based labellings Σ_0 and Σ'_0, two references r_x and r'_x such that:

- $\Gamma, L_{pc} \vdash e_0 \rightsquigarrow {}^{e'_0}/_{e''_0} : T$, $\omega = \mathsf{When}^?_\preceq(T, \Gamma(x))$, and $e' = e'_0, \mathsf{Wrap}(\omega, x = e''_0)$ (1) - (hyp.1) + (hyp.6)
- $r \vdash \langle \mu, \Sigma, e'_0 \rangle \Downarrow \langle \mu_0, \Sigma_f, v_f \rangle$, $r_x = \mathsf{Scope}(\mu_0, r, x)$, $\mu_f = \mu_0[r_x \cdot m_x \mapsto v_f]$, and $\mu_0, r \vDash \omega$ (2) - (hyp.2) + (hyp.6)
- $r \vdash \langle \mu', \Sigma', e'_0 \rangle \Downarrow \langle \mu'_0, \Sigma'_f, v'_f \rangle$, $r'_x = \mathsf{Scope}(\mu'_0, r, x)$, $\mu'_0 = \mu'_0[r'_x \cdot m_x \mapsto v'_f]$, and $\mu'_f, r \vDash \omega$ (3) - (hyp.3) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0$, $\Gamma, r \Vdash \mu_0 \sim_\sigma \mu'_0$, and:

$$\forall_{(\dot\tau_0, \omega_0) \in T}\ lev(\dot\tau_0) \sqsubseteq \sigma \Rightarrow (\mu_0, r \vDash \omega_0 \Leftrightarrow \mu'_0, r \vDash \omega_0)\ \wedge\ (\mu_0, r \vDash \omega_0 \Rightarrow v_f = v'_f)$$

(5) - (hyp.4) + (hyp.5) + (1) + (2) + (3) + **ih**

- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(6)** - (2) + (3) + (5)
- $\forall_{\hat\mu, \hat r}\ \hat\mu, \hat r \vDash \omega \Leftrightarrow \exists_{(\dot\tau_0, \omega_0) \in T}\ \dot\tau_0 \preceq \Gamma(x) \wedge \hat\mu, \hat r \vDash \omega_0$ (7) - (1)
- $\mu_0, r \vDash \omega \Leftrightarrow \exists_{(\dot\tau_0, \omega_0) \in T}\ \dot\tau_0 \preceq \Gamma(x) \wedge \mu_0, r \vDash \omega_0$ (8) - (7)
- $\exists_{(\dot\tau_0, \omega_0) \in T}\ \dot\tau_0 \preceq \Gamma(x) \wedge \mu_0, r \vDash \omega_0$ (9) - (2) + (8)
- $lev(\Gamma(x)) \sqsubseteq \sigma \Rightarrow \exists_{(\dot\tau_0, \omega_0) \in T}\ lev(\dot\tau_0) \sqsubseteq \sigma \wedge \mu_0, r \vDash \omega_0$ (10) - (9)
- $lev(\Gamma(x)) \sqsubseteq \sigma \Rightarrow v_f = v'_f$ (11) - (5) + (10)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(12)** - (2) + (3) + (5) + (11) + Indistinguishable Variable Assignment (Lemma B.5)

[Property Look-up] Suppose $e = e_0[e_1, P]^j$ for two expressions e_0 and e_1 (hyp.6). It follows that there are four memories μ_0, μ_1, μ'_0, and μ'_1, two type-based labelling Σ_0 and Σ'_0, four references r_0, $\hat r$, r'_0, and $\hat r'$ and two strings m_1 and m'_1, such that:

- $\Gamma, L_{pc} \vdash e_i \rightsquigarrow {}^{e'_i}/_{e''_i} : T_i$ for $i = 0, 1$, $e' = e'_0, e'_1, \$v_j = e''_0[e''_1]$, and:

$$T = \left(\pi_{\mathtt{type}}(\lceil^? (T_0, P, e''_1))\right)^{lev(T_0) \oplus_\sqcup lev(T_1)}$$

(1) - (hyp.1) + (hyp.6)

- $r \vdash \langle \mu, \Sigma, e'_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, r_0 \rangle$, $r \vdash \langle \mu_0, \Sigma_0, e'_1 \rangle \Downarrow \langle \mu_1, \Sigma_f, m_1 \rangle$, $\hat r = \mathsf{Proto}(\mu_1, r_0, m_1)$, $\hat r \neq \mathsf{null} \Rightarrow v_f = \mu_f(\hat r \cdot m_1)$, and $\hat r = \mathsf{null} \Rightarrow v = \mathsf{undefined}$ (2) - (hyp.2) + (hyp.6)
- $r \vdash \langle \mu', \Sigma', e'_0 \rangle \Downarrow \langle \mu'_0, \Sigma'_0, r'_0 \rangle$ and $r \vdash \langle \mu'_0, \Sigma'_0, e'_1 \rangle \Downarrow \langle \mu'_1, \Sigma'_f, m'_1 \rangle$ $\hat r' = \mathsf{Proto}(\mu'_f, r'_0, m'_1)$, $\hat r' \neq \mathsf{null} \Rightarrow v'_f = \mu'_f(\hat r' \cdot m'_1)$, and $\hat r = \mathsf{null} \Rightarrow v_f = \mathsf{undefined}$ (3) - (hyp.3) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0$, $\Gamma, r \Vdash \mu_0 \sim_\sigma \mu'_0$, and:

$$\forall_{(\dot\tau_0, \omega_0) \in T_0}\ lev(\dot\tau_0) \sqsubseteq \sigma \Rightarrow (\mu_0, r \vDash \omega_0 \Leftrightarrow \mu'_0, r \vDash \omega_0) \wedge (\mu_0, r \vDash \omega_0 \Rightarrow r_0 = r'_0)$$

(4) - (hyp.4) + (hyp.5) + (1) + (2) + (3) + **ih**

- $\mu_1, \Sigma_1 \sim_\sigma \mu'_1, \Sigma'_1$, $\Gamma, r \Vdash \mu_1 \sim_\sigma \mu'_1$, and:

$$\forall_{(\dot\tau_1, \omega_1) \in T_1}\ lev(\dot\tau_1) \sqsubseteq \sigma \Rightarrow (\mu_1, r \vDash \omega_1 \Leftrightarrow \mu'_1, r \vDash \omega_1) \wedge (\mu_1, r \vDash \omega_1 \Rightarrow m_1 = m'_1)$$

(5) - (1) + (2) + (3) + (4) + **ih**

- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ and $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(6)** - (1) + (2) + (3) + (5)

It remains to prove that:

$$\forall_{(\dot{\tau},\omega)\in T}\ lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow (\mu_f, r \vDash \omega \Leftrightarrow \mu'_f, r \vDash \omega) \wedge (\mu_f, r \vDash \omega \Rightarrow v_f = v'_f)$$

Suppose that $(\dot{\tau}, \omega) \in T$ (hyp.7.1), $lev(\dot{\tau}) \sqsubseteq \sigma$ (hyp.7.2), and $\mu_f, r \vDash \omega$ (hyp.7.3). It follows that there are $(\dot{\tau}_0, \omega_0), (\dot{\tau}'_0, \omega'_0) \in T_0$, $(\dot{\tau}_1, \omega_1) \in T_1$ and $p \in \texttt{Str}$ such that:

- $\omega \equiv \omega_0 \wedge \omega_1 \wedge \omega'_0 \wedge \omega_p$ and $\dot{\tau} = (\pi_{\texttt{type}}(\Rsh\ (\dot{\tau}'_0, p)))^{lev(\dot{\tau}_0)\sqcup lev(\dot{\tau}_1)}$, where:
  $\omega_p = \begin{cases} e''_0 \in \{p\} & \text{if } p \in dom(\dot{\tau}'_0) \\ \neg(e''_0 \in dom(\dot{\tau}'_0) \cap P) & \text{otherwise} \end{cases}$ (7) - (hyp.7.1)
- $\mu_f, r \vDash \omega_0$, $\mu_f, r \vDash \omega_1$, $\mu_f, r \vDash \omega'_0$, and $\mu_f, r \vDash \omega_{lu}$ (8) - (hyp.7.3) + (7)
- $\dot{\tau}'_0 = \dot{\tau}_0$ and $\omega'_0 = \omega_0$ (9) - (1) + (7) + Incompatible Assertions
- $\dot{\tau} = (\pi_{\texttt{type}}(\Rsh\ (\dot{\tau}_0, p)))^{lev(\dot{\tau}_0)\sqcup lev(\dot{\tau}_1)}$ and $\omega = \omega_0 \wedge \omega_1 \wedge \omega_p$, where:
  $\omega_p = \begin{cases} e''_0 \in \{p\} & \text{if } p \in dom(\dot{\tau}_0) \\ \neg(e''_0 \in dom(\dot{\tau}_0) \cap P) & \text{otherwise} \end{cases}$ (10) - (7) + (9)
- $lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqcup lev(\pi_{\texttt{type}}(\Rsh\ (\dot{\tau}_0, p))) \sqsubseteq \sigma$ (11) - (hyp.7.2) + (10)
- $\mu_f, r \vDash \omega_0 \Leftrightarrow \mu_0, r \vDash \omega_0$, $\mu_f, r \vDash \omega_1 \Leftrightarrow \mu_1, r \vDash \omega_1$, and $\mu_f, r \vDash \omega_p \Leftrightarrow \mu_1, r \vDash \omega_p$
  (12) - (1) + (2) + Invariance of Dynamic Assertions
- $\mu'_f, r \vDash \omega_0 \Leftrightarrow \mu'_0, r \vDash \omega_0$, $\mu'_f, r \vDash \omega_1 \Leftrightarrow \mu'_1, r \vDash \omega_1$, and $\mu'_f, r \vDash \omega_p \Leftrightarrow \mu'_1, r \vDash \omega_p$
  (13) - (1) + (3) + Invariance of Dynamic Assertions
- $\mu'_0, r \vDash \omega_0$ and $r_0 = r'_0$ (14) - (4) + (8) + (11) + (12)
- $\mu'_1, r \vDash \omega_1$ and $m_1 = m'_1$ (15) - (5) + (8) + (11) + (12)
- $\mu'_f, r \vDash \omega_0$ and $\mu'_f, r \vDash \omega_1$ (16) - (13)-(15)
- $\mu_1, r \vDash \omega_p \Rightarrow (p \in dom(\dot{\tau}_0) \wedge m_1 = p) \vee (p \notin dom(\dot{\tau}_0) \wedge m_1 \notin dom(\dot{\tau}_0))$ (17) - (2) + (7)
- $(p \in dom(\dot{\tau}_0) \wedge m_1 = p) \vee (p \notin dom(\dot{\tau}_0) \wedge m_1 \notin dom(\dot{\tau}_0))$ (18) - (8) + (12) + (17)
- $\Rsh\ (\dot{\tau}_0, m_1) = \Rsh\ (\dot{\tau}_0, p)$ (19) - (7) + (18)
- $\mu_0, r \vDash \omega_0 \Rightarrow \Sigma_0(r_0) \preceq \dot{\tau}_0$ (20) - (hyp.7.1) + (1) + (2) + Well-Labelled Memory
- $\mu'_0, r \vDash \omega_0 \Rightarrow \Sigma'_0(r'_0) \preceq \dot{\tau}_0$ (21) - (hyp.7.1) + (1) + (3) + Well-Labelled Memory
- $\mu_0, r \vDash \omega_0 \Rightarrow \Sigma_0(r_0) \curlyvee \Sigma'_0(r'_0) \preceq \dot{\tau}_0$ (22) - (4) + (11) + (20) + (21)
- $\mu_f, r \vDash \omega_0 \Rightarrow \Sigma_f(r_0) \curlyvee \Sigma'_f(r'_0) \preceq \dot{\tau}_0$
  (23) - (2) + (3) + (22) + Invariance of Dynamic Assertions
- $\Sigma_f(r_0) = \Sigma'_f(r_0) \preceq \dot{\tau}_0$ (24) - (4) + (8) + (11) + (14) + (23)
- $\lfloor \Sigma_f(r_0) \rfloor = \lfloor \Sigma'_f(r'_0) \rfloor = \lfloor \dot{\tau}_0 \rfloor$ (25) - (24)
- $\Rsh\ (\dot{\tau}_0, p) = \Rsh\ (\dot{\tau}_0, m_1) = \Rsh\ (\Sigma_f(r_0), m_1) = \Rsh\ (\Sigma'_f(r'_0), m'_1)$ (26) - (19) + (24)
- $lev(\pi_{\texttt{type}}(\Rsh\ (\Sigma_f(r_0), m_1))) = lev(\pi_{\texttt{type}}(\Rsh\ (\Sigma'_f(r'_0), m'_1))) \sqsubseteq \sigma$ (27) - (11) + (26)
- $lev(\pi_{\texttt{type}}(\Rsh\ (\Sigma_f(r_0), m_1))) \sqcup lev(\Sigma_f(r_0)) \sqsubseteq \sigma$ (28) - (11) + (25)
- $\hat{r} = \hat{r}'$ and $\hat{r} \neq \mathsf{null} \Rightarrow lev(\Sigma_f(\hat{r})) = lev(\Sigma'_f(\hat{r}')) \sqsubseteq \sigma$
  (29) - (2) + (3) + (6) + (28) + Prototype-Chain Indistinguishability (Lemma B.2)

We consider two cases: $\hat{r} \neq \mathsf{null}$ or $\hat{r} = \mathsf{null}$. Suppose $\hat{r} \neq \mathsf{null}$ (hyp.8):

- $\hat{r}' \neq \mathsf{null}$ and $lev(\Sigma_f(\hat{r})) = lev(\Sigma'_f(\hat{r}')) \sqsubseteq \sigma$ (30) - (hyp.8) + (29)
- $\Rsh\ (\Sigma_f(r_0), m_1) = \Rsh\ (\Sigma_f(\hat{r}), m_1)$ (31) - (1) + Well-Typed Prototype Chains (Lemma B.1)
- $\Rsh\ (\Sigma'_f(r'_0), m_1) = \Rsh\ (\Sigma'_f(\hat{r}'), m_1)$ (32) - (2) + Well-Typed Prototype Chains (Lemma B.1)

- $lev(\pi_{\texttt{type}}(\Pi\ (\Sigma_f(\hat{r}), m_1))) = lev(\pi_{\texttt{type}}(\Pi\ (\Sigma'_f(\hat{r}), m_1))) \sqsubseteq \sigma$ (33) - (27) + (31) + (32)
- $v_f = v'_f$ **(34)** - (hyp.8) + (2) + (3) + (6) + (15) + (29) + (30) + (33)

Suppose $\hat{r} = \mathsf{null}$ (hyp.8):

- $\hat{r}' = \mathsf{null}$ (35) - (hyp.8) + (29)
- $v_f = v'_f = \mathsf{undefined}$ **(36)** - (hyp.8) + (2) + (3) + (35)

[Membership Testing] This case is similar to the previous case. Therefore, the proof is omitted.

[Property Assignment] Suppose $e = e_0[e_1, P] = e_2$ for three expressions e_0, e_1, and e_2 (hyp.6). It follows that there are four memories μ_0, μ_1, μ'_0, and μ'_1, two type-based labelling Σ_0 and Σ'_0, four references r_0, $\hat{r}$, r'_0, and $\hat{r}'$ and two strings m_1 and m'_1, such that:

- $\Gamma, L_{pc} \vdash e_i \rightsquigarrow {}^{e'_i}/_{e''_i} : T_i$, $LT_P = \Pi^? (T_0, P, e''_1)$, $L_P = \pi_{\texttt{lev}}(LT)$, $T_P = \pi_{\texttt{type}}(LT)$, $T = T_2$, $\hat{\omega}_0 = \mathsf{When}^?_{\prec}(T_2, T_P)$, $\hat{\omega}_1 = \mathsf{When}^?_{\sqsubseteq}(L_{pc} \oplus_{\sqcup} lev(T_0) \oplus_{\sqcup} lev(T_1), L_P)$, and $e = e'_0, e'_1, e'_2, \mathsf{Wrap}(\omega_0 \wedge \omega_1, e''_0[e''_1] = e''_2)$ (1) - (hyp.1) + (hyp.6)
- $r \vdash \langle \mu, \Sigma, e'_0 \rangle \Downarrow \langle \mu_0, \Sigma_0, r_0 \rangle$, $r \vdash \langle \mu_0, \Sigma_0, e'_1 \rangle \Downarrow \langle \mu_1, \Sigma_1, m_1 \rangle$, $r \vdash \langle \mu_1, \Sigma_1, e'_2 \rangle \Downarrow \langle \mu_2, \Sigma_f, v_2 \rangle$, $\mu_f = \mu_2[r_0 \cdot m_1 \mapsto v_2]$, $\mu_2, r \vDash \hat{\omega}_0$, and $\mu_2, r \vDash \hat{\omega}_1$ (2) - (hyp.2) + (hyp.6)
- $r \vdash \langle \mu', \Sigma', e'_0 \rangle \Downarrow \langle \mu'_0, \Sigma'_0, r'_0 \rangle$, $r \vdash \langle \mu'_0, \Sigma'_0, e'_1 \rangle \Downarrow \langle \mu'_1, \Sigma'_1, m'_1 \rangle$, $r \vdash \langle \mu'_1, \Sigma'_1, e'_2 \rangle \Downarrow \langle \mu'_2, \Sigma'_2, v'_2 \rangle$, $\mu_f = \mu_2[r_0 \cdot m_1 \mapsto v_2]$, $\mu_2, r \vDash \hat{\omega}_0$, and $\mu_2, r \vDash \hat{\omega}_1$ (3) - (hyp.3) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0, \Gamma, r \Vdash \mu_0 \sim_\sigma \mu'_0$,

$$\forall_{(\dot{\tau}_0, \omega_0) \in T_0}\ lev(\dot{\tau}_0) \sqsubseteq \sigma \Rightarrow (\mu_0, r \vDash \omega_0 \Leftrightarrow \mu'_0, r \vDash \omega_0) \wedge (\mu_0, r \vDash \omega_0 \Rightarrow r_0 = r'_0)$$

(4) - (hyp.4) + (hyp.5) + (1) + (2) + (3) + **ih**

- $\mu_1, \Sigma_1 \sim_\sigma \mu'_1, \Sigma'_1, \Gamma, r \Vdash \mu_1 \sim_\sigma \mu'_1$,

$$\forall_{(\dot{\tau}_1, \omega_1) \in T_1}\ lev(\dot{\tau}_1) \sqsubseteq \sigma \Rightarrow (\mu_1, r \vDash \omega_1 \Leftrightarrow \mu'_1, r \vDash \omega_1) \wedge (\mu_1, r \vDash \omega_1 \Rightarrow m_1 = m'_1)$$

(5) - (1) + (2) + (3) + (4) + **ih**

- $\mu_2, \Sigma_2 \sim_\sigma \mu'_2, \Sigma'_2, \Gamma, r \Vdash \mu_2 \sim_\sigma \mu'_2$,

$$\forall_{(\dot{\tau}_2, \omega_2) \in T_2}\ lev(\dot{\tau}_2) \sqsubseteq \sigma \Rightarrow (\mu_2, r \vDash \omega_2 \Leftrightarrow \mu'_2, r \vDash \omega_2) \wedge (\mu_2, r \vDash \omega_2 \Rightarrow v_2 = v'_2)$$

(6) - (1) + (2) + (3) + (5) + **ih**

- $\forall_{\hat{\mu}, \hat{r}}\ \hat{\mu}, \hat{r} \vDash \hat{\omega}_1 \Leftrightarrow \exists_{(\dot{\tau}_0, \omega_0) \in T_0, (\dot{\tau}_1, \omega_1) \in T_1, (\dot{\tau}_2, \omega_2) \in T_2, (\sigma_{pc}, \omega_{pc}) \in L_{pc}, p \in \texttt{Str}}$
$\hat{\mu}, \hat{r} \vDash \omega_0 \wedge \hat{\mu}, \hat{r} \vDash \omega_1 \wedge \hat{\mu}, \hat{r} \vDash \omega_2 \wedge \hat{\mu}, \hat{r} \vDash \omega_{pc} \wedge \hat{\mu}, \hat{r} \vDash \omega_p \wedge$
$lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqcup \sigma_{pc} \sqsubseteq \pi_{\texttt{lev}}(\Pi\ (\dot{\tau}_0, p)) \wedge \dot{\tau}_2 \preceq \pi_{\texttt{type}}(\Pi\ (\dot{\tau}_0, p))$

  where:
  $\omega_p = \begin{cases} e''_0 \in \{p\} & \text{if } p \in dom(\dot{\tau}_0) \\ \neg(e''_0 \in dom(\dot{\tau}_0) \cap P) & \text{otherwise} \end{cases}$ (8) - (1)
- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(9)** - (2) + (3) + (6)

From (2) and (8), we conclude that there are $\dot{\tau}_0, \omega_0) \in T_0$, $(\dot{\tau}_1, \omega_1) \in T_1$, $(\dot{\tau}_2, \omega_2) \in T_2$, $(\sigma_{pc}, \omega_{pc}) \in L_{pc}$, and $p \in \texttt{Str}$, such that:

- $\mu_2, r \vDash \omega_0$, $\mu_2, r \vDash \omega_1$, $\mu_2, r \vDash \omega_2$, $\mu_2, r \vDash \omega_{pc}$, and $\mu_2, r \vDash \omega_p$ (10) - (2) + (8)
- $lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqcup \sigma_{pc} \sqsubseteq \pi_{\texttt{lev}}(\Pi\ (\dot{\tau}_0, p))$ and $\dot{\tau}_2 \preceq \pi_{\texttt{type}}(\Pi\ (\dot{\tau}_0, p))$ (11) - (2) + (8)
- $\mu_2, r \vDash \omega_0 \Leftrightarrow \mu_0, r \vDash \omega_0$, $\mu_2, r \vDash \omega_1 \Leftrightarrow \mu_1, r \vDash \omega_1$, and $\mu_2, r \vDash \omega_p \Leftrightarrow \mu_1, r \vDash \omega_p$
(12) - (1) + (2) + Invariance of Dynamic Assertions

We consider two different cases: either $lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqcup lev\dot{\tau}_2 \sqcup \sigma_{pc} \sqsubseteq \sigma$ or $lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqcup lev\dot{\tau}_2 \sqcup \sigma_{pc} \not\sqsubseteq \sigma$. Suppose $lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqcup lev\dot{\tau}_2 \sqcup \sigma_{pc} \sqsubseteq \sigma$ (hyp.7). We conclude that:

- $\mu'_0, r \vDash \omega_0$ and $r_0 = r'_0$ (13) - (hyp.7) + (4) + (10) - (12)
- $\mu'_1, r \vDash \omega_1$ and $m_1 = m'_1$ (14) - (hyp.7) + (5) + (10) - (12)
- $\mu'_2, r \vDash \omega_2$ and $v_2 = v'_2$ (15) - (hyp.7) + (6) + (10) - (12)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ (16) - (6) + (13)-(15)

Suppose $lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqcup lev\dot{\tau}_2 \sqcup \sigma_{pc} \not\sqsubseteq \sigma$ (hyp.7). We conclude that:

- $\mu_f \restriction^{\Sigma_f,\sigma} = \mu_2 \restriction^{\Sigma_2,\sigma}$ (17) - (hyp.7) + (2)
- $\mu'_f \restriction^{\Sigma'_f,\sigma} = \mu'_2 \restriction^{\Sigma'_2,\sigma}$ (18) - (hyp.7) + (3)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(19)** - (6) + (17) + (18)

[Property Deletion] Suppose $e = \mathsf{delete}^{i,P}\ e_0[e_1]$ for some expression e_0, property p, and index i (hyp.6). It follows that there are four memories μ_0, μ_1, μ'_0, and μ'_1, and two type-based labellings Σ_0 and Σ_1 such that:

- $\Gamma, L_{pc} \vdash e_0 \rightsquigarrow {}^{e'_0}/_{e''_0} : T_0$, $\Gamma, L_{pc} \vdash e_1 \rightsquigarrow {}^{e'_1}/_{e''_1} : T_1$, $T = \{(\mathsf{PRIM}^\perp, \mathsf{true})\}$, $e' = e'_0, e'_1, \mathsf{Wrap}(\omega, \$v_i = \mathsf{delete}\ e''_0[e''_1])$, and $\omega = \mathsf{When}^?_\sqsubseteq(lev(T_0) \oplus_\sqcup lev(T_1), \pi_{\mathtt{lev}}(\Rsh^?\ (T_0, P, e''_1)))$ (1) - (hyp.1) + (hyp.6)
- $r \vdash \langle \mu, \Sigma, e'_0 \rangle \Downarrow \langle \mu_0, \Sigma_f, r_0 \rangle$, $r \vdash \langle \mu_0, \Sigma_0, e'_1 \rangle \Downarrow \langle \mu_1, \Sigma_f, m_1 \rangle$ $\mu_f = \mu_1 \left[r_0 \mapsto \mu_1(r_0)|_{dom(\mu_1(r_0)\setminus\{m_1\})} \right]$, and $\mu_1, r \vDash \omega$, $v_f = \mathsf{true}$ (2) - (hyp.2) + (hyp.6)
- $r \vdash \langle \mu', \Sigma', e'_0 \rangle \Downarrow \langle \mu'_0, \Sigma'_f, r'_0 \rangle$, $r \vdash \langle \mu'_0, \Sigma'_0, e'_1 \rangle \Downarrow \langle \mu'_1, \Sigma'_f, m'_1 \rangle$ $\mu'_f = \mu'_1 \left[r'_0 \mapsto \mu'_1(r'_0)|_{dom(\mu'_1(r'_0)\setminus\{m'_1\})} \right]$, and $\mu'_1, r \vDash \omega$, $v'_f = \mathsf{true}$ (3) - (hyp.3) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0$, $\Gamma, r \Vdash \mu_0 \sim_\sigma \mu'_0$, and

$$\forall_{(\dot{\tau}_0,\omega_0)\in T_0}\ lev(\dot{\tau}_0) \sqsubseteq \sigma \Rightarrow (\mu_0, r \vDash \omega_0 \Leftrightarrow \mu'_0, r \vDash \omega_0) \wedge (\mu_0, r \vDash \omega_0 \Rightarrow r_0 = r'_0)$$

(4) - (hyp.4) + (hyp.5) + (1) + (2) + (3) + **ih**

- $\mu_1, \Sigma_1 \sim_\sigma \mu'_1, \Sigma'_1$, $\Gamma, r \Vdash \mu_1 \sim_\sigma \mu'_1$, and

$$\forall_{(\dot{\tau}_1,\omega_1)\in T_1}\ lev(\dot{\tau}_1) \sqsubseteq \sigma \Rightarrow (\mu_1, r \vDash \omega_1 \Leftrightarrow \mu'_1, r \vDash \omega_1) \wedge (\mu_1, r \vDash \omega_1 \Rightarrow m_1 = m'_1)$$

(5) - (1) + (2) + (3) + (4) + **ih**

- $\Gamma, r \Vdash \mu_f \sim_\sigma \mu'_f$ **(6)** - (2)-(5)
- $v_f = v'_f = \mathsf{true}$ (7) - (2) + (3)
- $\mu_f, r \vDash \mathsf{true} \Rightarrow v_f = v'_f$ (8) - (7)
- $\mu_f, r \vDash \mathsf{true} \Leftrightarrow \mu'_f, r \vDash \mathsf{true}$ (9) - tautology
- $\forall_{(\dot{\tau},\omega)\in T}\ lev(\dot{\tau}) \sqsubseteq \sigma \Rightarrow (\mu_f, r \vDash \omega \Leftrightarrow \mu'_f, r \vDash \omega) \wedge (\mu_f, r \vDash \omega \Rightarrow v_f = v'_f)$ **(10)** - (1) + (8) + (9)
- $\forall_{\hat{\mu},\hat{r}}\ \hat{\mu}, \hat{r} \vDash \omega \Leftrightarrow \exists_{(\dot{\tau}_0,\omega_0)\in T_0,(\dot{\tau}_1,\omega_1)\in T_1, p\in\mathtt{Str}}$ $lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \preceq \pi_{\mathtt{lev}}(\Rsh\ (\dot{\tau}_0, p))\ \wedge\ \hat{\mu}, \hat{r} \vDash \omega_0\ \wedge\ \hat{\mu}, \hat{r} \vDash \omega_1\ \wedge\ \hat{\mu}, \hat{r} \vDash \omega_p$ (11) - (1)
- $\mu_1, r \vDash \omega \Leftrightarrow$ $\exists_{(\dot{\tau}_0,\omega_0)\in T_0,(\dot{\tau}_1,\omega_1)\in T_1, p\in\mathtt{Str}}\ lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqsubseteq \pi_{\mathtt{lev}}(\Rsh\ (\dot{\tau}_0, p)) \wedge \mu_1, r \vDash \omega_0 \wedge \mu_1, r \vDash \omega_1 \wedge \mu_1, r \vDash \omega_p$ (12) - (11)
- There are $(\dot{\tau}_0, \omega_0) \in T_0$, $(\dot{\tau}_1, \omega_1) \in T_1$, and string $p \in \mathtt{Str}$ such that:

$$lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqsubseteq \pi_{\mathtt{lev}}(\Rsh\ (\dot{\tau}_0, p))\ \wedge\ \mu_1, r \vDash \omega_0\ \wedge\ \mu_1, r \vDash \omega_1\ \wedge\ \mu_1, r \vDash \omega_p$$

(13) - (2) + (11)

We consider two cases: $lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqsubseteq \sigma$ and $lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \not\sqsubseteq \sigma$. Suppose $lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \sqsubseteq \sigma$ (hyp.7). It follows that:

- $r_0 = r'_0$ and $m_1 = m'_1$ (14) - (hyp.7) + (4) + (5) + (13)

- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(15)** - (2) + (3) + (5) + (14)

Suppose $lev(\dot{\tau}_0) \sqcup lev(\dot{\tau}_1) \not\sqsubseteq \sigma$ (hyp.7). It follows that:

- $\pi_{\texttt{lev}}(\lceil^{\star} (\Sigma_f(r_0), m_1)) \sqcap \pi_{\texttt{lev}}(\lceil^{\star} (\Sigma'_f(r'_0), m'_1)) \not\sqsubseteq \sigma$ (16) - (hyp.7) + (5) + (13)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$ **(17)** - (2) + (3) + (4) + (19)

The proofs of the remaining cases are done in a similar way. □

Appendix C

Proofs of Chapter 6

Lemma 6.1 - Confinement for the Extensible Monitor

Proof: Given an API $\mathsf{API}_{IF} = \langle \mathcal{S}, \mathcal{S}_{lab}, \mathcal{P}, \mathcal{P}_{lab}, \mathcal{R}_{IF}, \sim_{api} \rangle$, the hypothesis of the lemma are the following:

- $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \mid \nu, \Xi \rangle \Downarrow_{IF}^{\mathsf{API}_{IF}} \langle \mu', v', \Sigma', \sigma' \mid \nu', \Xi' \rangle$ (hyp.1)
- $\sigma_{pc} \not\sqsubseteq \sigma$ (hyp.2)
- API_{IF} is confined (hyp.3)

We have to prove that:

- $\mu_f, \Sigma_f \sim_\sigma \mu', \Sigma'$,
- $\nu, \Xi \sim_{api}^{\sigma} \nu'_f, \Xi'_f$ where $\sim_{api} = \mathsf{API}_{IF}.\mathsf{equality}$,
- $\sigma' \not\sqsubseteq \sigma$.

As in the case of the proof of confinement for the Core JavaScript monitor (Theorem 4.1), we proceed by induction on the derivation of (hyp.1). Instead of re-examining the whole monitor, we only consider the rules: [External Property Look-up] and [External Method Call]. In the following, we use $\mathcal{R}_{IF}$ for $\mathsf{API}_{IF}.\mathsf{Reg}$.

[External Property Look-up Literal] We conclude that $e = e_0[e_1]^\alpha$ (hyp.4) and that there is a reference r_0 and a string m_1 such that $\langle r_0, m_1 \rangle \in dom(\mathcal{R}_{IF})$ (hyp.5) and:

- $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \mid \nu, \Xi \rangle \Downarrow_{IF}^{\mathsf{API}_{IF}} \langle \mu_0, r_0, \Sigma_0, \sigma_0 \mid \nu_0, \Xi_0 \rangle$ (1) - (hyp.1) + (hyp.4)
- $r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0 \mid \nu_0, \Xi_0 \rangle \Downarrow_{IF}^{\mathsf{API}_{IF}} \langle \mu_1, m_1, \Sigma_1, \sigma_1 \mid \nu_1, \Xi_1 \rangle$ (2) - (hyp.1) + (hyp.4)
- $(\mathsf{pg}, \mathsf{pg}_{lab}) = \mathcal{R}_{IF}(r_0, m_1)$ (3) - (hyp.5)
- $\langle \nu_1, r_0 :: m_1 \rangle^\alpha$ pg $\langle \nu', v' \rangle^\beta$ and $\langle \Xi_1, \sigma_0 :: \sigma_1 \rangle^\beta$ pg_{lab} $\langle \Xi', \sigma \rangle$ (4) - (hyp.1) + (hyp.3) + (hyp.4)
- $\mu, \Sigma \sim_\sigma \mu_0, \Sigma_0$ and $\nu, \Xi \sim_{api}^{\sigma} \nu_0, \Xi_0$ (5) - (hyp.2) + (hyp.3) + (1) + **ih**
- $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$ and $\nu_0, \Xi_0 \sim_{api}^{\sigma} \nu_1, \Xi_1$ (6) - (hyp.2) + (hyp.3) + (2) + **ih**
- $\mu_1, \Sigma_1 \sim_\sigma \mu', \Sigma'$ and $\nu_2, \Xi_2 \sim_{api}^{\sigma} \nu', \Xi'$ (7) - (hyp.2) + (4) + *Cofinement for APIs* (Definition 6.3)
- $\mu, \Sigma \sim_\sigma \mu', \Sigma'$ and $\nu, \Xi \sim_{api}^{\sigma} \nu', \Xi'$ **(8)** - (5) - (7) + *Transitivity of the Low-Equality*

[External Method Call] Suppose that $e = e_0[e_1](e_2)^\alpha$ (hyp.4). We conclude that there is a reference r_0 and a string m_1 such that $\langle r_0, m_1 \rangle \in dom(\mathcal{R}_{IF})$ (hyp.5) and:

- $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \mid \nu, \Xi \rangle \Downarrow_{IF}^{\mathsf{API}_{IF}} \langle \mu_0, r_0, \Sigma_0, \sigma_0 \mid \nu_0, \Xi_0 \rangle$ (1) - (hyp.1) + (hyp.4)
- $r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0 \mid \nu_0, \Xi_0 \rangle \Downarrow_{IF}^{\mathsf{API}_{IF}} \langle \mu_1, m_1, \Sigma_1, \sigma_1 \mid \nu_1, \Xi_1 \rangle$ (2) - (hyp.1) + (hyp.4)
- $r, \sigma_{pc} \vdash \langle \mu_1, e_2, \Sigma_1 \mid \nu_1, \Xi_1 \rangle \Downarrow_{IF}^{\mathsf{API}_{IF}} \langle \mu_2, v_2, \Sigma_2, \sigma_2 \mid \nu_2, \Xi_2 \rangle$ (3) - (hyp.1) + (hyp.4)
- $(\mathsf{pg}, \mathsf{pg}_{lab}) = \mathcal{R}_{IF}(r_0, m_1)$ (4) - (hyp.5)

- $\langle \nu_2, r_0 :: m_1 :: v_2 \rangle^\alpha$ pg $\langle \nu', v' \rangle^\beta$ and $\langle \Xi_2, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^\beta$ pg_{lab} $\langle \Xi', \sigma' \rangle$
  (5) - (hyp.1) + (hyp.3) + (hyp.4)
- $\mu, \Sigma \sim_\sigma \mu_0, \Sigma_0$ and $\nu, \Xi \sim^\sigma_{api} \nu_0, \Xi_0$ (6) - (hyp.2) + (hyp.3) + (1) + **ih**
- $\mu_0, \Sigma_0 \sim_\sigma \mu_1, \Sigma_1$ and $\nu_0, \Xi_0 \sim^\sigma_{api} \nu_1, \Xi_1$ (7) - (hyp.2) + (hyp.3) + (2) + **ih**
- $\mu_1, \Sigma_1 \sim_\sigma \mu_2, \Sigma_2$ and $\nu_1, \Xi_1 \sim^\sigma_{api} \nu_2, \Xi_2$ (8) - (hyp.2) + (hyp.3) + (2) + **ih**
- $\mu_2, \Sigma_2 \sim_\sigma \mu', \Sigma'$ and $\nu_2, \Xi_2 \sim^\sigma_{api} \nu', \Xi'$
  (9) - (hyp.2) + (hyp.3) + (4) + (5) + *Cofinement for APIs* (Definition 6.3)
- $\mu, \Sigma \sim_\sigma \mu', \Sigma'$ and $\nu, \Xi \sim^\sigma_{api} \nu', \Xi'$ **(10)** - (6) - (9) + *Transitivity of the Low-Equality*

□

Theorem 6.1 - Noninterference for the Extensible Monitor

Proof: Suppose that $\mathsf{API}_{IF} = \langle \mathcal{S}, \mathcal{S}_{lab}, \mathcal{P}, \mathcal{P}_{lab}, \mathcal{R}_{IF}, \sim_{api} \rangle$, $\sim_{api} = \mathsf{API}_{IF}.\mathsf{equality}$ and $\mathcal{R}_{IF} = \mathsf{API}_{IF}.\mathsf{Reg}$. We restate the hypotheses of the theorem:

- $\mu, \Sigma \sim_\sigma \mu', \Sigma'$ (hyp.1),
- $\nu, \Xi \sim^\sigma_{api} \nu', \Xi'$ where (hyp.2),
- $r, \sigma_{pc} \vdash \langle \mu, e, \Sigma \mid \nu, \Xi \rangle \Downarrow^{\mathsf{API}_{IF}}_{IF} \langle \mu_f, v_f, \Sigma_f, \sigma_f \mid \nu_f, \Xi_f \rangle$ (hyp.3),
- $r, \sigma_{pc} \vdash \langle \mu', e, \Sigma' \mid \nu', \Xi' \rangle \Downarrow^{\mathsf{API}_{IF}}_{IF} \langle \mu'_f, v'_f, \Sigma'_f, \sigma'_f \mid \nu'_f, \Xi'_f \rangle$(hyp.4).

We have to prove that:

- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$,
- $\nu_f, \Xi_f \sim_\sigma \nu'_f, \Xi'_f$,
- $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$.

As in the case of the proof of noninterference for the Core JavaScript monitor, we proceed by induction on the derivation of (hyp.3).

[External Property Look-up Literal] We conclude that $e = e_0[e_1]^\alpha$ (hyp.5) and that there exist a reference r_0 and a string m_1 such that $\langle r_0, m_1 \rangle \in dom(\mathcal{R}_{IF})$ (hyp.6) and:

- $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \mid \nu, \Xi \rangle \Downarrow^{\mathsf{API}_{IF}}_{IF} \langle \mu_0, r_0, \Sigma_0, \sigma_0 \mid \nu_0, \Xi_0 \rangle$ and $r, \sigma_{pc} \vdash \langle \mu', e_0, \Sigma' \mid \nu', \Xi' \rangle \Downarrow^{\mathsf{API}_{IF}}_{IF}$ $\langle \mu'_0, r'_0, \Sigma'_0, \sigma'_0 \mid \nu'_0, \Xi'_0 \rangle$ (1) - (hyp.1) + (hyp.3) + (hyp.4)
- $r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0 \mid \nu_0, \Xi_0 \rangle \Downarrow^{\mathsf{API}_{IF}}_{IF} \langle \mu_1, m_1, \Sigma_1, \sigma_1 \mid \nu_1, \Xi_1 \rangle$ and $r, \sigma_{pc} \vdash \langle \mu'_0, e_1, \Sigma'_0 \mid \nu'_0, \Xi'_0 \rangle \Downarrow^{\mathsf{API}_{IF}}_{IF}$ $\langle \mu'_1, m'_1, \Sigma'_1, \sigma'_1 \mid \nu'_1, \Xi'_1 \rangle$ (2) - (hyp.1) + (hyp.3) + (hyp.4)
- $(\mathsf{pg}, \mathsf{pg}_{lab}) = \mathcal{R}_{IF}(r_0, m_1)$ (3) - (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0$, $\nu_0, \Xi_0 \sim^\sigma_{api} \nu'_0, \Xi'_0$, and $r_0, \sigma_0 \sim_\sigma r'_0, \sigma'_0$
  (4) - (hyp.1) + (hyp.2) + (1) + **ih**
- $\mu_1, \Sigma_1 \sim_\sigma \mu'_1, \Sigma'_1$, $\nu_1, \Xi_1 \sim^\sigma_{api} \nu'_1, \Xi'_1$, and $m_1, \sigma_1 \sim_\sigma m'_1, \sigma'_1$ (5) - (2) + (4) + **ih**
- $\langle \nu_1, r_0 :: m_1 \rangle^\alpha$ pg $\langle \nu_f, v_f \rangle^\beta$ and $\langle \Xi_1, \sigma_0 :: \sigma_1 \rangle^\beta$ pg_{lab} $\langle \Xi_f, \sigma_f \rangle$
  (6) - (hyp.3) - (hyp.6) + (3)

There are two cases to consider: $\sigma_0 \sqcup \sigma_1 \sqsubseteq \sigma$ or $\sigma_0 \sqcup \sigma_1 \not\sqsubseteq \sigma$. Suppose that $\sigma_0 \sqcup \sigma_1 \sqsubseteq \sigma$ (hyp.7):

- $r_0 = r'_0$, $m_1 = m'_1$, $\sigma'_0 = \sigma_0$, and $\sigma_1 = \sigma'_1$ (7) - (hyp.7) + (4) + (5)
- $(\mathsf{pg}, \mathsf{pg}_{lab}) = \mathcal{R}_{IF}(r'_0, m'_1)$ (8) - (3) + (7)

- $\langle \nu'_1, r_0 :: m_1 \rangle^\alpha$ pg $\langle \nu'_f, v'_f \rangle^\beta$ and $\langle \Xi'_1, \sigma_0 :: \sigma_1 \rangle^\beta$ pg_{lab} $\langle \Xi'_f, \sigma'_f \rangle$ (9) - (hyp.4) + (2) + (8)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$, $\nu_f, \Xi_f \sim^\sigma_{api} \nu'_f, \Xi'_f$, and $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$ **(10)** - (5) + (7)-(9) + *Noninterferent API* (Definition 6.4)

Suppose that $\sigma_0 \sqcup \sigma_1 \not\sqsubseteq \sigma$ (hyp.7):

- $\sigma'_0 \sqcup \sigma'_1 \not\sqsubseteq \sigma$ (11) - (hyp.7) + (4) + (5)
- $\mu_1, \Sigma_1 \sim_\sigma \mu_f, \Sigma_f$, $\nu_1, \Xi_1 \sim^\sigma_{api} \nu_f, \Xi_f$, and $\sigma_f \not\sqsubseteq \sigma$ (12) - (hyp.7) + (6) + *Cofinement for APIs* (Definition 6.3)
- $\mu'_1, \Sigma'_1 \sim_\sigma \mu'_f, \Sigma'_f$, $\nu'_1, \Xi'_1 \sim^\sigma_{api} \nu'_f, \Xi'_f$, and $\sigma'_f \not\sqsubseteq \sigma$ (13) - (hyp.4) + (11) + *Cofinement* (Lemma 6.1)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$, $\nu_f, \Xi_f \sim^\sigma_{api} \nu'_f, \Xi'_f$, and $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$ **(14)** - (5) + (12) + (13)

[External Method Call] We conclude that $e = e_0[e_1](e_2)^\alpha$ (hyp.5) and that there exist a reference r_0 and a string m_1 such that $\langle r_0, m_1 \rangle \in dom(\mathcal{R}_{IF})$ (hyp.6) and:

- $r, \sigma_{pc} \vdash \langle \mu, e_0, \Sigma \mid \nu, \Xi \rangle \Downarrow^{\mathsf{API}_{IF}}_{IF} \langle \mu_0, r_0, \Sigma_0, \sigma_0 \mid \nu_0, \Xi_0 \rangle$ and $r, \sigma_{pc} \vdash \langle \mu', e_0, \Sigma' \mid \nu', \Xi' \rangle \Downarrow^{\mathsf{API}_{IF}}_{IF} \langle \mu'_0, r'_0, \Sigma'_0, \sigma'_0 \mid \nu'_0, \Xi'_0 \rangle$ (1) - (hyp.1) + (hyp.3) + (hyp.4)
- $r, \sigma_{pc} \vdash \langle \mu_0, e_1, \Sigma_0 \mid \nu_0, \Xi_0 \rangle \Downarrow^{\mathsf{API}_{IF}}_{IF} \langle \mu_1, m_1, \Sigma_1, \sigma_1 \mid \nu_1, \Xi_1 \rangle$ and $r, \sigma_{pc} \vdash \langle \mu'_0, e_1, \Sigma'_0 \mid \nu'_0, \Xi'_0 \rangle \Downarrow^{\mathsf{API}_{IF}}_{IF} \langle \mu'_1, m'_1, \Sigma'_1, \sigma'_1 \mid \nu'_1, \Xi'_1 \rangle$ (2) - (hyp.1) + (hyp.3) + (hyp.4)
- $r, \sigma_{pc} \vdash \langle \mu_1, e_2, \Sigma_1 \mid \nu_1, \Xi_1 \rangle \Downarrow^{\mathsf{API}_{IF}}_{IF} \langle \mu_2, v_2, \Sigma_2, \sigma_2 \mid \nu_2, \Xi_2 \rangle$ and $r, \sigma_{pc} \vdash \langle \mu'_1, e_2, \Sigma'_1 \mid \nu'_1, \Xi'_1 \rangle \Downarrow^{\mathsf{API}_{IF}}_{IF} \langle \mu'_2, v'_2, \Sigma'_2, \sigma'_2 \mid \nu'_2, \Xi'_2 \rangle$ (3) - (hyp.1) + (hyp.3) + (hyp.4)
- $(\mathsf{pg}, \mathsf{pg}_{lab}) = \mathcal{R}_{IF}(r_0, m_1)$ (4) - (hyp.5) + (hyp.6)
- $\mu_0, \Sigma_0 \sim_\sigma \mu'_0, \Sigma'_0$, $\nu_0, \Xi_0 \sim^\sigma_{api} \nu'_0, \Xi'_0$, and $r_0, \sigma_0 \sim_\sigma r'_0, \sigma'_0$ (5) - (hyp.1) + (hyp.2) + (1) + **ih**
- $\mu_1, \Sigma_1 \sim_\sigma \mu'_1, \Sigma'_1$, $\nu_1, \Xi_1 \sim^\sigma_{api} \nu'_1, \Xi'_1$, and $m_1, \sigma_1 \sim_\sigma m'_1, \sigma'_1$ (6) - (2) + (5) + **ih**
- $\mu_2, \Sigma_2 \sim_\sigma \mu'_2, \Sigma'_2$, $\nu_2, \Xi_2 \sim^\sigma_{api} \nu'_2, \Xi'_2$, and $v_2, \sigma_2 \sim_\sigma v'_2, \sigma'_2$ (7) - (3) + (6) + **ih**
- $\langle \nu_2, r_0 :: m_1 :: v_2 \rangle^\alpha$ pg $\langle \nu_f, v_f \rangle^\beta$ and $\langle \Xi_2, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^\beta$ pg_{lab} $\langle \Xi_f, \sigma_f \rangle$ (8) - (hyp.3) + (hyp.4) + (hyp.5) + (4)

There are two cases to consider: $\sigma_0 \sqcup \sigma_1 \sqsubseteq \sigma$ or $\sigma_0 \sqcup \sigma_1 \not\sqsubseteq \sigma$. Suppose that $\sigma_0 \sqcup \sigma_1 \sqsubseteq \sigma$ (hyp.7):

- $r_0 = r'_0$, $m_1 = m'_1$, $\sigma'_0 = \sigma_0$, and $\sigma_1 = \sigma'_1$ (9) - (hyp.7) + (5) + (6)
- $(\mathsf{pg}, \mathsf{pg}_{lab}) = \mathcal{R}_{IF}(r'_0, m'_1)$ (10) - (4) + (9)
- $r_0 :: m_1 :: v_2, \sigma_0 :: \sigma_1 :: \sigma_2 \sim_\sigma r_0 :: m_1 :: v'_2, \sigma_0 :: \sigma_1 :: \sigma'_2$ (11) - (7) + (9)
- $\langle \nu'_2, r_0 :: m_1 :: v'_2 \rangle^\alpha$ pg $\langle \nu'_f, v'_f \rangle^\beta$ and $\langle \Xi'_1, \sigma_0 :: \sigma_1 :: \sigma'_2 \rangle^\beta$ pg_{lab} $\langle \Xi'_f, \sigma'_f \rangle$ (12) - (hyp.4) + (10)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$, $\nu_f, \Xi_f \sim^\sigma_{api} \nu'_f, \Xi'_f$, and $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$ **(13)** - (7) + (8) + (12) + *Noninterferent API* (Definition 6.4)

Suppose that $\sigma_0 \sqcup \sigma_1 \not\sqsubseteq \sigma$ (hyp.7):

- $\sigma'_0 \sqcup \sigma'_1 \not\sqsubseteq \sigma$ (14) - (hyp.7) + (5) + (6)
- $\mu_2, \Sigma_2 \sim_\sigma \mu_f, \Sigma_f$, $\nu_2, \Xi_2 \sim^\sigma_{api} \nu_f, \Xi_f$, and $\sigma_f \not\sqsubseteq \sigma$ (15) - (hyp.7) + (8) + *Cofinement for APIs* (Definition 6.3)
- $\mu'_2, \Sigma'_2 \sim_\sigma \mu'_f, \Sigma'_f$, $\nu'_2, \Xi'_2 \sim^\sigma_{api} \nu'_f, \Xi'_f$, and $\sigma'_f \not\sqsubseteq \sigma$ (16) - (hyp.4) + (14) + *Cofinement* (Lemma 6.1)
- $\mu_f, \Sigma_f \sim_\sigma \mu'_f, \Sigma'_f$, $\nu_f, \Xi_f \sim^\sigma_{api} \nu'_f, \Xi'_f$, and $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$ **(17)** - (7) + (15) + (16)

□

Appendix D

Proofs of Chapter 7

D.1 Noninterference - Basic DOM API

In this section we give the proofs of:

- Lemma 7.1 - Well-labelling Preservation
- Lemma 7.2 - Confinement of the Monitored Core DOM API
- Theorem 7.1 - Noninterference of the Monitored Core DOM API

Lemma 7.1 - Well-labelling Preservation

Proof: For every $(\mathsf{pg}, \mathsf{pg}_{lab}) \in dom(\mathcal{R}_{IF}^{DOM})$, we have to prove that given a forest f well-labelled by Ξ and a sequence of values $\overrightarrow{v}$ labelled by a sequence of levels $\overrightarrow{\sigma}$, such that:

- $\langle f, \overrightarrow{v} \rangle^{\alpha}\ \mathsf{pg}\ \langle f', v' \rangle^{\beta}$ (hyp.1)
- $\langle \Xi, \overrightarrow{\sigma} \rangle^{\beta}\ \mathsf{pg}_{lab}\ \langle \Xi', \sigma' \rangle$ (hyp.2)

Then, it holds that: f' is well-labelled by Ξ'.

We proceed by case analysis. We only consider the plugins that can change the memory.

[Store] Given that:

- $\langle f, r :: \texttt{"storeValue"} :: v \rangle\ \mathsf{store}\ \langle f', v \rangle^{(r)}$ (hyp.1)
- $\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^{(r)}\ \mathsf{store}_{lab}\ \langle \Xi', \sigma' \rangle$ (hyp.2)

we have to prove that: f' is well-labelled by Ξ'. Letting $\overrightarrow{r} = f(r).\mathsf{children}$, $m = f(r).\mathsf{tag}$, $\hat{r} = f(r).\mathsf{parent}$, we conclude that:

- $f' = f\,[r \mapsto \langle m, v, \hat{r}, \overrightarrow{r} \rangle]$ (1) - (hyp.1)
- $\sigma' = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqcup \Xi(r).\mathsf{node}$, (2) - (hyp.2)
- $\Xi' = \Xi\,[r \mapsto \langle \Xi(r).\mathsf{node}, \sigma', \Xi(r).\mathsf{pos}, \Xi(r).\mathsf{struct} \rangle]$ (3) - (hyp.2)
- $\Xi'(r).\mathsf{node} \sqsubseteq \Xi'(r).\mathsf{value}$ **(4)** - (2) + (3)

Remark: the labellings of the other nodes do not change. Hence, they do not have to be verified. The other components of the labelling of the node pointed to by r do not change. Therefore, they do not have to be verified.

[Remove] Given that:

- $\langle f, r :: \texttt{"removeChild"} :: r' \rangle\ \mathsf{remove}\ \langle f', r' \rangle^{(r,r')}$ (hyp.1)
- $\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^{(r,r')}\ \mathsf{remove}_{lab}\ \langle \Xi, \sigma' \rangle$ (hyp.2)

we have to prove that: f' is well-labelled by Ξ. The removal of a node from the list of children of the node pointed to by r ($f(r)$) does not compromise the fact that the position levels of the children of $f(r)$ are monotonically increasing. Therefore, there is nothing to prove.

[Append] Given that:

- $\langle f, r :: \texttt{"appendChild"} :: r' \rangle$ append $\langle f', r' \rangle^{(r,r',r'')}$ (hyp.1)
- $\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^{(r,r',r'')}$ append_{lab} $\langle \Xi, \sigma' \rangle$ (hyp.2)

we have to prove that: f' is well-labelled by Ξ. Letting $m = f(r).\mathsf{tag}$, $v = f(r).\mathsf{value}$, $\hat{r} = f(r).\mathsf{parent}$, $\overrightarrow{r} = f(r).\mathsf{children}$, $m' = f(r').\mathsf{tag}$, $v' = f(r').\mathsf{value}$, $\hat{r}' = f(r').\mathsf{parent}$, and $\overrightarrow{r}' = f(r').\mathsf{children}$, and $i = |f(r).\mathsf{children}| - 1$, we conclude that:

- $f(r').\mathsf{parent} = \mathsf{null}$ (1) - (hyp.1)
- $f' = f[r \mapsto \langle m, v, \hat{r}, \overrightarrow{r} :: r' \rangle, r' \mapsto \langle m', v', r, \overrightarrow{r}' \rangle]$ (2) - (hyp.1)
- $\overrightarrow{r} \neq \varepsilon \Rightarrow \Xi(\overrightarrow{r}.\mathsf{last}).\mathsf{pos} \sqsubseteq \Xi(r').\mathsf{pos}$ (3) - (hyp.1) + (hyp.2)
- $\Xi(r).\mathsf{node} \sqsubseteq \Xi(r').\mathsf{node}$ **(4)** - (hyp.1) + (hyp.2)
- $\Xi(f'(r).\mathsf{children}(i)).\mathsf{pos} \sqsubseteq \Xi(f'(r).\mathsf{children}(i+1)).\mathsf{pos}$ **(5)** - (2) + (3)

[Node Creation] Given that:

- $\langle f, \#doc :: \texttt{"createElement"} :: m \rangle^{(\sigma_n,\sigma_p,\sigma_s)}$ new $\langle f', r \rangle^{(r,\sigma_n,\sigma_p,\sigma_s)}$ (hyp.1)
- $\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^{(r,\sigma_n,\sigma_p,\sigma_s)}$ new_{lab} $\langle \Xi', \sigma' \rangle$ (hyp.2)

we have to prove that f' is well-labelled by Ξ'. We conclude that:

- $r = \mathsf{fresh}_{DOM}(\sigma_n)$ (1) - (hyp.1)
- $f' = f\,[r \mapsto \langle m, \mathsf{null}, \mathsf{null}, \varepsilon \rangle]$ (2) - (hyp.1) + (1)
- $\Xi' = \Xi\,[r \mapsto \langle \sigma_n, \sigma_n, \sigma_p, \sigma_s \rangle]$ (3) - (hyp.2) + (1)
- $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \sigma_n \sqsubseteq \sigma_p \sqcap \sigma_s$ **(4)** - (hyp.2)

□

Lemma 7.2 - Confinement of the Core DOM API

Proof: We proceed by case analysis. We only consider the monitored plugins that can change the memory.

[Store] Given that:

- $\langle f, r :: \texttt{"storeValue"} :: v \rangle$ store $\langle f', v \rangle^{(r)}$ (hyp.1)
- $\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^{(r)}$ store_{lab} $\langle \Xi', \sigma' \rangle$ (hyp.2)
- $\sigma_0 \sqcup \sigma_1 \not\sqsubseteq \sigma$ (hyp.3)

we have to prove that: $f, \Xi \sim^{\sigma}_{DOM} f', \Xi'$ and $\sigma' \not\sqsubseteq \sigma$. Letting $\overrightarrow{r} = f(r).\mathsf{children}$, $m = f(r).\mathsf{tag}$, $v' = f(r).\mathsf{value}$, $\hat{r} = f(r).\mathsf{parent}$, we conclude that:

- $f' = f\,[r \mapsto \langle m, v, \hat{r}, \overrightarrow{r} \rangle]$ (1) - (hyp.1)
- $\sigma' = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqcup \Xi(r).\mathsf{node}$, (2) - (hyp.2)
- $\Xi' = \Xi\,[r \mapsto \langle \Xi(r).\mathsf{node}, \sigma', \Xi(r).\mathsf{pos}, \Xi(r).\mathsf{struct} \rangle]$ (3) - (hyp.2)
- $\sigma_0 \sqcup \sigma_1 \sqsubseteq \Xi(r).\mathsf{value}$ (4) - (hyp.2)

- $\Xi(r).\mathsf{value} \not\sqsubseteq \sigma$ (5) - (hyp.3) + (4)
- $(r, v', \Xi(r).\mathsf{value}) \notin f \restriction^{\Xi,\sigma}$ (6) - (5)
- $\sigma' \not\sqsubseteq \sigma$ **(7)** - (hyp.3) + (4)
- $(r, v, \Xi'(r).\mathsf{value}) \notin f' \restriction^{\Xi',\sigma}$ (8) - (1) - (3) + (7)
- $f, \Xi \sim^{\sigma}_{DOM} f', \Xi'$ **(9)** - (1) + (6) + (8)

[Remove] Given that:

- $\langle f, r :: \texttt{"removeChild"} :: r' \rangle$ remove $\langle f', r' \rangle^{(r,r')}$ (hyp.1)
- $\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^{(r,r')}$ remove_{lab} $\langle \Xi, \sigma' \rangle$ (hyp.2)
- $\sigma_0 \sqcup \sigma_1 \not\sqsubseteq \sigma$ (hyp.3)

we have to prove that: $f, \Xi \sim^{\sigma}_{DOM} f', \Xi$ and $\sigma' \not\sqsubseteq \sigma$. Letting $\overrightarrow{r} = f(r).\mathsf{children}$, $m = f(r).\mathsf{tag}$, $v = f(r).\mathsf{value}$, $\hat{r} = f(r).\mathsf{parent}$, $m' = f(r').\mathsf{tag}$, $v' = f(r').\mathsf{value}$, $\overrightarrow{r}' = f(r').\mathsf{children}$, we conclude that:

- There is an integer i such that: $\overrightarrow{r}(i) = r'$ (1) - (hyp.1)
- $f' = f[r \mapsto \langle m, v, \hat{r}, \mathsf{Shift}_{\mathsf{L}}(\overrightarrow{r}, i) \rangle, r' \mapsto \langle m', v', \mathsf{null}, \overrightarrow{r}' \rangle]$ (2) - (hyp.1) + (1)
- $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \Xi(r).\mathsf{struct} \sqcap \Xi(r').\mathsf{pos}$ (3) - (hyp.2)
- $\Xi(r).\mathsf{struct} \sqcap \Xi(r').\mathsf{pos} \not\sqsubseteq \sigma$ (4) - (hyp.3) + (3)
- $(r, i, r') \notin f \restriction^{\Xi,\sigma}$ and $(r', \mathsf{null}) \notin f' \restriction^{\Xi,\sigma}$ (5) - (hyp.1) + (hyp.2) + (4)
- $(r, |\overrightarrow{r}|) \notin f \restriction^{\Xi,\sigma}$ and $(r, |\mathsf{Shift}_{\mathsf{L}}(\overrightarrow{r}, i)|) \notin f' \restriction^{\Xi,\sigma}$ (6) - (hyp.1) + (hyp.2) + (4)
- $\forall_{i<j<|\overrightarrow{r}|}\ \Xi(\overrightarrow{r}(j)).\mathsf{pos} \not\sqsubseteq \sigma$ (7) - (4) + *Well-labelled Forest*
- $\forall_{i<j<|\overrightarrow{r}|}\ (r, j, \overrightarrow{r}(j)) \notin f \restriction^{\Xi,\sigma}$ (8) - (7)
- $\forall_{i\leq j<|\mathsf{Shift}_{\mathsf{L}}(\overrightarrow{r},i)|}\ (r, j, \mathsf{Shift}_{\mathsf{L}}(\overrightarrow{r}, i)(j)) \notin f' \restriction^{\Xi,\sigma}$ (9) - (4) + (7)
- $f \restriction^{\Xi,\sigma} = f' \restriction^{\Xi,\sigma}$ **(10)** - (5)+(6)+(8)+(9)
- $\sigma' = \Xi(r).\mathsf{pos}$ (11) - (hyp.2)
- $\sigma' \not\sqsubseteq \sigma$ **(12)** - (hyp.3) + (4)

[Append] Given that:

- $\langle f, r :: \texttt{"appendChild"} :: r' \rangle$ append $\langle f', r' \rangle^{(r,r',r'')}$ (hyp.1)
- $\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^{(r,r',r'')}$ append_{lab} $\langle \Xi, \sigma' \rangle$ (hyp.2)
- $\sigma_0 \sqcup \sigma_1 \not\sqsubseteq \sigma$ (hyp.3)

we have to prove that: $f, \Xi \sim^{\sigma}_{DOM} f', \Xi$ and $\sigma' \not\sqsubseteq \sigma$. Letting $\overrightarrow{r} = f(r).\mathsf{children}$, $m = f(r).\mathsf{tag}$, $v = f(r).\mathsf{value}$, $\hat{r} = f(r).\mathsf{parent}$, $m' = f(r').\mathsf{tag}$, $v' = f(r').\mathsf{value}$, $\overrightarrow{r}' = f(r').\mathsf{children}$, we conclude that:

- $f(r').\mathsf{parent} = \mathsf{null}$ (1) - (hyp.1)
- $f' = f[r \mapsto \langle m, v, \hat{r}, \overrightarrow{r} :: r' \rangle, r' \mapsto \langle m', v', r, \overrightarrow{r}' \rangle]$ (2) - (hyp.1)
- $\sigma_0 \sqcup \sigma_1 \sqsubseteq \Xi(r).\mathsf{struct} \sqcap \Xi(r').\mathsf{pos}$ (3) - (hyp.2)
- $\Xi(r).\mathsf{struct} \sqcap \Xi(r').\mathsf{pos} \not\sqsubseteq \sigma$ (4) - (hyp.3) + (3)
- $(r', \mathsf{null}) \notin f \restriction^{\Xi,\sigma}$ and $(r, i, r') \notin f' \restriction^{\Xi,\sigma}$, where $i = |\overrightarrow{r}|$ (5) - (4)

- $(r, |\overrightarrow{r}|) \notin f \restriction^{\Xi,\sigma}$ and $(r, |\overrightarrow{r}|+1) \notin f' \restriction^{\Xi,\sigma}$ (6) - (4)
- $f, \Xi \sim^{\sigma}_{DOM} f', \Xi$ **(7)** - (5) + (6)
- $\sigma' = \Xi.\mathsf{pos}(r')$ (8) - (hyp.2)
- $\sigma' \not\sqsubseteq \sigma$ **(9)** - (4) + (8)

[Node Creation] Given that:

- $\langle f, \#doc :: \texttt{"createElement"} :: m\rangle^{(\sigma_n,\sigma_p,\sigma_s)}\ \mathsf{new}\ \langle f', r\rangle^{(r,\sigma_n,\sigma_p,\sigma_s)}$ (hyp.1)
- $\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2\rangle^{(r,\sigma_n,\sigma_p,\sigma_s)}\ \mathsf{new}_{lab}\ \langle \Xi', \sigma'\rangle$ (hyp.2)
- $\sigma_0 \sqcup \sigma_1 \not\sqsubseteq \sigma$ (hyp.3)

we have to prove that: $f, \Xi \sim^{\sigma}_{DOM} f', \Xi'$ and $\sigma' \not\sqsubseteq \sigma$. We conclude that:

- $r = \mathsf{fresh}_{DOM}(\sigma_n)$ (1) - (hyp.1)
- $f' = f\,[r \mapsto \langle m, \mathsf{null}, \mathsf{null}, \varepsilon\rangle]$ (2) - (hyp.1) + (1)
- $\Xi' = \Xi\,[r \mapsto \langle \sigma_n, \sigma_n, \sigma_p, \sigma_s\rangle]$ (3) - (hyp.2) + (1)
- $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \sigma_n \sqsubseteq \sigma_p \sqcap \sigma_s$ (4) - (hyp.2)
- $\sigma_n \sqcap \sigma_p \sqcap \sigma_s \not\sqsubseteq \sigma$ (5) - (hyp.3) + (4)
- $f \restriction^{\Xi,\sigma} = f' \restriction^{\Xi',\sigma}$ **(6)** - (2) + (3) + (5)
- $\sigma' = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2$ (8) - (hyp.2)
- $\sigma' \not\sqsubseteq \sigma$ **(9)** - (hyp.3) + (8)

□

Theorem 7.1 - Noninterference of the Monitored Core DOM API

Proof: For every $(\mathsf{pg}, \mathsf{pg}_{lab}) \in dom(\mathcal{R}^{DOM}_{IF})$, we have to prove that given two forests f and f' labelled by Ξ and Ξ' and two sequences of values $\overrightarrow{v}$ and $\overrightarrow{v}'$ respectively labelled by two sequences of levels $\overrightarrow{\sigma}$ and $\overrightarrow{\sigma}'$ and such that:

- $\overrightarrow{v}, \overrightarrow{\sigma} \sim^{\sigma}_{DOM} \overrightarrow{v}, \overrightarrow{\sigma}'$ (hyp.1)
- $f, \Xi \sim^{\sigma}_{DOM} f', \Xi'$ (hyp.2)
- $\langle f, \overrightarrow{v}\rangle^{\alpha}\ \mathsf{pg}\ \langle f_f, v_f\rangle^{\beta}$ (hyp.3) and $\langle f', \overrightarrow{v}'\rangle^{\alpha}\ \mathsf{pg}\ \langle f'_f, v'_f\rangle^{\beta'}$ (hyp.4)
- $\langle \Xi, \overrightarrow{\sigma}\rangle^{\beta}\ \mathsf{pg}_{lab}\ \langle \Xi_f, \sigma_f\rangle$ (hyp.5) and $\langle \Xi', \overrightarrow{\sigma}'\rangle^{\beta'}\ \mathsf{pg}_{lab}\ \langle \Xi'_f, \sigma'_f\rangle$ (hyp.6)

Then, it holds that: $f_f, \Xi_f \sim^{\sigma}_{DOM} f'_f, \Xi'_f$ and $v_f, \sigma_f \sim_{\sigma} v'_f, \sigma'_f$. In order to prove that $v_f, \sigma_f \sim_{\sigma} v'_f, \sigma'_f$, we have to prove the following two implications: **(1)** $\sigma_f \sqsubseteq \sigma \Rightarrow v_f = v'_f \wedge \sigma_f = \sigma'_f \sqsubseteq \sigma$ and **(2)** $\sigma'_f \sqsubseteq \sigma \Rightarrow v_f = v'_f \wedge \sigma_f = \sigma'_f \sqsubseteq \sigma$. Since the proofs of **(1)** and **(2)** are identical, we only prove **(1)**. However, we cannot introduce at this level the hypothesis $\sigma_f \sqsubseteq \sigma$ because it cannot be used in the proof of $f_f, \Xi_f \sim^{\sigma}_{DOM} f'_f, \Xi'_f$. Therefore, we are obliged to introduce this hypothesis in every case. We now proceed by case analysis on the API methods in the range of $\mathcal{R}^{DOM}_{IF}$.

[Parent] Suppose $(\mathsf{pg}, \mathsf{pg}_{lab}) = (\mathsf{parent}, \mathsf{parent}_{lab})$ (hyp.7). We conclude that there are two node references r and r' such that:

- $\overrightarrow{v} = r :: \texttt{"parentNode"}$, $\overrightarrow{v}' = r' :: \texttt{"parentNode"}$, $\overrightarrow{\sigma} = \sigma_0 :: \sigma_1$, and $\overrightarrow{\sigma}' = \sigma'_0 :: \sigma'_1$ (1) - (hyp.3) - (hyp.7)

- $\sigma_f = \sigma_0 \sqcup \sigma_1 \sqcup \Xi(r).\mathsf{pos}$ and $\sigma'_f = \sigma'_0 \sqcup \sigma'_1 \sqcup \Xi'(r').\mathsf{pos}$ (2) - (hyp.3) - (hyp.7)
- $\sigma_0 \sqcap \sigma'_0 \sqsubseteq \sigma \Rightarrow r = r' \wedge \sigma_0 = \sigma'_0 \sqsubseteq \sigma$ (3) - (hyp.1) + (1)
- $\sigma_1 \sqcap \sigma'_1 \sqsubseteq \sigma \Rightarrow \sigma_1 = \sigma'_1 \sqsubseteq \sigma$ (4) - (hyp.1) + (1)
- $f_f = f$, $\Xi_f = \Xi$, $v_f = f(r).\mathsf{parent}$ (5) - (hyp.3) + (hyp.5) + (hyp.7)
- $f'_f = f'$, $\Xi'_f = \Xi'$, and $v'_f = f'(r').\mathsf{parent}$ (6) - (hyp.4) + (hyp.6) + (hyp.7)
- $f_f, \Xi_f \sim^{\sigma}_{DOM} f'_f, \Xi'_f$ **(7)** - (hyp.2) + (5) + (6)

In the following suppose that $\sigma_f \sqsubseteq \sigma$ (hyp.8):

- $\sigma_0 \sqsubseteq \sigma$, $\sigma_1 \sqsubseteq \sigma$, and $\Xi(r).\mathsf{pos} \sqsubseteq \sigma$ (8) - (hyp.8) + (2)
- $r = r'$, $\sigma_0 = \sigma'_0$, and $\sigma_1 = \sigma'_1$ (9) - (3) + (4) + (8)
- $f(r).\mathsf{parent} = f'(r').\mathsf{parent}$ and $\Xi(r).\mathsf{pos} = \Xi'(r').\mathsf{pos}$ (10) - (hyp.2) + (8) + (9)
- $v_f = v'_f$ and $\sigma_f = \sigma'_f$ **(11)** - (2) + (5) + (6) + (9) + (10)

[ITEM] Suppose $(\mathsf{pg}, \mathsf{pg}_{lab}) = (\mathsf{item}, \mathsf{item}_{lab})$ (hyp.7). We conclude that there are two node references r and r' such that:

- $\overrightarrow{v} = r :: i$, $\overrightarrow{v}' = r' :: j$, $\overrightarrow{\sigma} = \sigma_0 :: \sigma_1$, and $\overrightarrow{\sigma}' = \sigma'_0 :: \sigma'_1$ (1) - (hyp.3) - (hyp.7)
- $v_f = \hat{r} = f(r).\mathsf{children}(i) \neq \mathsf{null}$ and $v'_f = \hat{r}' = f'(r').\mathsf{children}(j) \neq \mathsf{null}$ (2) - (hyp.3) + (hyp.4) + (hyp.7)
- $\sigma_f = \sigma_0 \sqcup \sigma_1 \sqcup \Xi(\hat{r}).\mathsf{pos}$ and $\sigma'_f = \sigma'_0 \sqcup \sigma'_1 \sqcup \Xi'(\hat{r}').\mathsf{pos}$ (3) - (hyp.5) + (hyp.6) + (hyp.7) + (2)
- $f_f = f$ and $\Xi_f = \Xi$ (4) - (hyp.3) + (hyp.5) + (hyp.7)
- $f'_f = f'$ and $\Xi'_f = \Xi'$ (5) - (hyp.4) + (hyp.6) + (hyp.7)
- $f_f, \Xi_f \sim^{\sigma}_{DOM} f'_f, \Xi'_f$ **(6)** - (hyp.2) + (4) + (5)

In the following suppose that $\sigma_f \sqsubseteq \sigma$ (hyp.8):

- $\sigma_0 \sqcap \sigma'_0 \sqsubseteq \sigma \Rightarrow r = r' \wedge \sigma_0 = \sigma'_0 \sqsubseteq \sigma$ (7) - (hyp.1) + (1)
- $\sigma_1 \sqcap \sigma'_1 \sqsubseteq \sigma \Rightarrow i = j \wedge \sigma_1 = \sigma'_1 \sqsubseteq \sigma$ (8) - (hyp.1) + (1)
- $\sigma_0 \sqsubseteq \sigma$, $\sigma_1 \sqsubseteq \sigma$, and $\Xi(\hat{r}).\mathsf{pos} \sqsubseteq \sigma$ (9) - (hyp.7) + (3)
- $r = r'$, $\sigma_0 = \sigma'_0$, $i = j$, and $\sigma_1 = \sigma'_1$ (10) - (7) - (9)
- $\hat{r} = \hat{r}'$ and $\Xi(\hat{r}).\mathsf{pos} = \Xi'(\hat{r}').\mathsf{pos} \sqsubseteq \sigma$ (11) - (hyp.2) + (9) + (10)
- $v_f = v'_f$ and $\sigma_f = \sigma'_f$ **(12)** - (2) + (3) + (10) + (11)

[LENGTH] Suppose $(\mathsf{pg}, \mathsf{pg}_{lab}) = (\mathsf{length}, \mathsf{length}_{lab})$ (hyp.7). We conclude that there are two node references r and r' such that:

- $\overrightarrow{v} = r :: \texttt{"length"}$, $\overrightarrow{v}' = r' :: \texttt{"length"}$, $\overrightarrow{\sigma} = \sigma_0 :: \sigma_1$, and $\overrightarrow{\sigma}' = \sigma'_0 :: \sigma'_1$ (1) - (hyp.3) - (hyp.7)
- $v_f = |f(r).\mathsf{children}|$ and $v'_f = |f'(r').\mathsf{children}|$ (2) - (hyp.3) + (hyp.4) + (hyp.7)
- $\sigma_f = \sigma_0 \sqcup \sigma_1 \sqcup \Xi(r).\mathsf{struct}$ and $\sigma'_f = \sigma'_0 \sqcup \sigma'_1 \sqcup \Xi'(r').\mathsf{struct}$ (3) - (hyp.5) + (hyp.6) + (hyp.7)
- $f_f = f$ and $\Xi_f = \Xi$ (4) - (hyp.3) + (hyp.5) + (hyp.7)
- $f'_f = f'$ and $\Xi'_f = \Xi'$ (5) - (hyp.4) + (hyp.6) + (hyp.7)
- $f_f, \Xi_f \sim^{\sigma}_{DOM} f'_f, \Xi'_f$ **(6)** - (hyp.2) + (4) + (5)

In the following suppose that $\sigma_f \sqsubseteq \sigma$ (hyp.8):

- $\sigma_0 \sqcap \sigma'_0 \sqsubseteq \sigma \Rightarrow r = r' \wedge \sigma_0 = \sigma'_0 \sqsubseteq \sigma$ (7) - (hyp.1) + (1)
- $\sigma_1 \sqcap \sigma'_1 \sqsubseteq \sigma \Rightarrow \sigma_1 = \sigma'_1 \sqsubseteq \sigma$ (8) - (hyp.1) + (1)

- $\sigma_0 \sqsubseteq \sigma$, $\sigma_1 \sqsubseteq \sigma$, and $\Xi(r).\mathsf{struct} \sqsubseteq \sigma$ (9) - (hyp.8) + (3)
- $r = r'$, $\sigma_0 = \sigma'_0$, and $\sigma_1 = \sigma'_1$ (10) - (7) - (9)
- $|f(r).\mathsf{children}| = |f'(r').\mathsf{children}|$ and $\Xi(r).\mathsf{struct} = \Xi'(r').\mathsf{struct} \sqsubseteq \sigma$ (11) - (hyp.2) + (9) + (10)
- $v_f = v'_f$ and $\sigma_f = \sigma'_f$ **(12)** - (2) + (3) + (10) + (11)

[VALUE] Suppose $(\mathsf{pg}, \mathsf{pg}_{lab}) = (\mathsf{value}, \mathsf{value}_{lab})$ (hyp.7). We conclude that there are two node references r and r' such that:

- $\overrightarrow{v} = r :: \texttt{"nodeValue"}$, $\overrightarrow{v}' = r' :: \texttt{"nodeValue"}$, $\overrightarrow{\sigma} = \sigma_0 :: \sigma_1$, and $\overrightarrow{\sigma}' = \sigma'_0 :: \sigma'_1$ (1) - (hyp.3) - (hyp.7)
- $v_f = f(r).\mathsf{value}$ and $v'_f = f'(r').\mathsf{value}$ (2) - (hyp.3) + (hyp.4) + (hyp.7)
- $\sigma_f = \sigma_0 \sqcup \sigma_1 \sqcup \Xi(r).\mathsf{value}$ and $\sigma'_f = \sigma'_0 \sqcup \sigma'_1 \sqcup \Xi'(r').\mathsf{value}$ (3) - (hyp.5) + (hyp.6) + (hyp.7)
- $f_f = f$ and $\Xi_f = \Xi$ (4) - (hyp.3) + (hyp.5) + (hyp.7)
- $f'_f = f'$ and $\Xi'_f = \Xi'$ (5) - (hyp.4) + (hyp.6) + (hyp.7)
- $f_f, \Xi_f \sim^{\sigma}_{DOM} f'_f, \Xi'_f$ **(6)** - (hyp.2) + (4) + (5)

In the following suppose that $\sigma_f \sqsubseteq \sigma$ (hyp.8):

- $\sigma_0 \sqcap \sigma'_0 \sqsubseteq \sigma \Rightarrow r = r' \wedge \sigma_0 = \sigma'_0 \sqsubseteq \sigma$ (7) - (hyp.1) + (1)
- $\sigma_1 \sqcap \sigma'_1 \sqsubseteq \sigma \Rightarrow \sigma_1 = \sigma'_1 \sqsubseteq \sigma$ (8) - (hyp.1) + (1)
- $\sigma_0 \sqsubseteq \sigma$, $\sigma_1 \sqsubseteq \sigma$, and $\Xi(r).\mathsf{value} \sqsubseteq \sigma$ (9) - (hyp.8) + (3)
- $r = r'$, $\sigma_0 = \sigma'_0$, and $\sigma_1 = \sigma'_1$ (10) - (7) - (9)
- $f(r).\mathsf{value} = f'(r').\mathsf{value}$ and $\Xi(r).\mathsf{value} = \Xi'(r').\mathsf{value} \sqsubseteq \sigma$ (11) - (hyp.2) + (9) + (10)
- $v_f = v'_f$ and $\sigma_f = \sigma'_f$ **(12)** - (2) + (3) + (10) + (11)

[NEW] Suppose $(\mathsf{pg}, \mathsf{pg}_{lab}) = (\mathsf{new}, \mathsf{new}_{lab})$ (hyp.7). We conclude that there are two strings m and m', nine security levels σ_0, σ_1, σ_2, σ'_0, σ'_1, σ'_2, σ_n, σ_p, and σ_s, two references r and r', and an index i, such that:

- $\overrightarrow{v} = \#doc :: \texttt{"createElement"} :: m$, $\overrightarrow{v}' = \#doc :: \texttt{"createElement"} :: m'$, $\overrightarrow{\sigma} = \sigma_0 :: \sigma_1 :: \sigma_2$, and $\overrightarrow{\sigma}' = \sigma'_0 :: \sigma'_1 :: \sigma'_2$ (1) - (hyp.3) - (hyp.7)
- $v_f = r = \mathsf{fresh}_{DOM}(\sigma_n)$ and $v'_f = \mathsf{fresh}_{DOM}(\sigma_n)$ (2) - (hyp.3) + (hyp.4) + (hyp.7)
- $\sigma_f = \sigma_n$ and $\sigma'_f = \sigma_n$ (3) - (hyp.5) + (hyp.6) + (hyp.7)
- $\sigma_n \sqsubseteq \sigma_p \sqcap \sigma_s$ (4) - (hyp.5) + (hyp.6) + (hyp.7)
- $f_f = f\,[r \mapsto \langle m, \mathsf{null}, \mathsf{null}, \varepsilon\rangle]$ and $\Xi_f = \Xi\,[r \mapsto \langle \sigma_n, \sigma_n, \sigma_p, \sigma_s\rangle]$ (5) - (hyp.3) + (hyp.5) + (hyp.7)
- $f'_f = f'\,[r' \mapsto \langle m', \mathsf{null}, \mathsf{null}, \varepsilon\rangle]$ and $\Xi'_f = \Xi'\,[r' \mapsto \langle \sigma_n, \sigma_n, \sigma_p, \sigma_s\rangle]$ (6) - (hyp.4) + (hyp.6) + (hyp.7)

In the following suppose that $\sigma_0 \sqcup \sigma_1 \not\sqsubseteq \sigma$ (hyp.8):

- $\sigma'_f \not\sqsubseteq \sigma$ (7) - (hyp.1) + (hyp.8) + (3)
- $f, \Xi \sim^{\sigma}_{DOM} f_f, \Xi_f$ (8) - (hyp.3) + (hyp.5) + (hyp.8) + Confinement (Lemma 7.2)
- $f', \Xi' \sim^{\sigma}_{DOM} f'_f, \Xi'_f$ (9) - (hyp.4) + (hyp.6) + (7) + Confinement (Lemma 7.2)
- $f_f, \Xi_f \sim^{\sigma}_{DOM} f'_f, \Xi'_f$ **(10)** - (hyp.2) + (8) + (9) + Reflexivity and Symmetry of $\sim^{\sigma}_{DOM}$

In the following suppose that $\sigma_f \not\sqsubseteq \sigma$ (hyp.8):

- $\sigma_0 \sqcap \sigma'_0 \sqsubseteq \sigma \Rightarrow \sigma_0 = \sigma'_0 \sqsubseteq \sigma$ (11) - (hyp.1) + (1)
- $\sigma_1 \sqcap \sigma'_1 \sqsubseteq \sigma \Rightarrow \sigma_1 = \sigma'_1 \sqsubseteq \sigma$ (12) - (hyp.1) + (1)

- $\sigma_2 \sqcap \sigma_2' \sqsubseteq \sigma \Rightarrow m = m' \wedge \sigma_2 = \sigma_2' \sqsubseteq \sigma$ (13) - (hyp.1) + (1)
- $\sigma_0 \sqsubseteq \sigma$, $\sigma_1 \sqsubseteq \sigma$, and $\sigma_2 \sqsubseteq \sigma$ (14) - (hyp.8) + (3)
- $\sigma_0 = \sigma_0'$, $\sigma_1 = \sigma_1'$, $\sigma_2 = \sigma_2'$, and $m = m'$ (15) - (11) - (14)
- $r = r'$ (16) - (hyp.2) + (2) + *Parametric Allocation*
- $f_f(r).\mathsf{tag} = f_f'(r').\mathsf{tag}$, $f_f(r).\mathsf{value} = f_f'(r').\mathsf{value}$, $f_f(r).\mathsf{children} = f_f'(r').\mathsf{children}$ (17) - (5) + (6) + (15) + (16)
- $\Xi_f(r).\mathsf{node} = \Xi_f'(r').\mathsf{node}$, $\Xi_f(r).\mathsf{pos} = \Xi_f'(r').\mathsf{pos}$, $\Xi_f(r).\mathsf{value} = \Xi_f'(r').\mathsf{value}$, and $\Xi_f(r).\mathsf{struct} = \Xi_f'(r').\mathsf{struct}$ (18) - (5) + (6) + (15)
- $f_f, \Xi_f \sim_{DOM}^{\sigma} f_f', \Xi_f'$ and $v_f = v_f'$ **(19)** - (hyp.2) + (17) + (18)

[STORE] Suppose $(\mathsf{pg}, \mathsf{pg}_{lab}) = (\mathsf{store}, \mathsf{store}_{lab})$ (hyp.7). We conclude that there are two references r and r' and two values v and v' such that:

- $\overrightarrow{v} = r :: \texttt{"storeValue"} :: v$, $\overrightarrow{v}' = r' :: \texttt{"storeValue"} :: v'$, $\overrightarrow{\sigma} = \sigma_0 :: \sigma_1 :: \sigma_2$, and $\overrightarrow{\sigma}' = \sigma_0' :: \sigma_1' :: \sigma_2'$ (1) - (hyp.3) - (hyp.7)
- $v_f = v$ and $v_f' = v'$ (2) - (hyp.3) + (hyp.4) + (hyp.7)
- $\sigma_f = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqcup \Sigma(r).\mathsf{node}$ and $\sigma_f' = \sigma_0' \sqcup \sigma_1' \sqcup \sigma_2' \sqcup \Sigma(r).\mathsf{node}$ (3) - (hyp.5) + (hyp.6) + (hyp.7)
- $\sigma_f \sqsubseteq \Xi(r).\mathsf{node}$ and $\sigma_f' \sqsubseteq \Xi'(r').\mathsf{node}$ (4) - (hyp.5) + (hyp.6) + (hyp.7)
- $\sigma_0 \sqcup \sigma_1 \sqsubseteq \Xi(r).\mathsf{value}$ and $\sigma_0' \sqcup \sigma_1' \sqsubseteq \Xi'(r').\mathsf{value}$ (5) - (hyp.5) + (hyp.6) + (hyp.7)
- $f_f = f\,[r \mapsto \langle f(r).\mathsf{tag}, v, f(r).\mathsf{parent}, f(r).\mathsf{children}\rangle]$ and $\Xi_f = \Xi\,[r \mapsto \langle \Xi(r).\mathsf{node}, \sigma_f, \Xi(r).\mathsf{pos}, \Xi(r).\mathsf{struct}\rangle]$ (6) - (hyp.3) + (hyp.5) + (hyp.7)
- $f_f' = f'\,[r' \mapsto \langle f'(r').\mathsf{tag}, v', f'(r').\mathsf{parent}, f'(r').\mathsf{children}\rangle]$ and $\Xi_f' = \Xi'\left[r' \mapsto \langle \Xi(r').\mathsf{node}, \sigma_f', \Xi(r').\mathsf{pos}, \Xi(r').\mathsf{struct}\rangle\right]$ (7) - (hyp.4) + (hyp.6) + (hyp.7)

In the following suppose that $\sigma_0 \sqcup \sigma_1 \not\sqsubseteq \sigma$ (hyp.8):

- $\sigma_0' \sqcup \sigma_1' \not\sqsubseteq \sigma$ (8) - (hyp.1) + (hyp.8)
- $f, \Xi \sim_{DOM}^{\sigma} f_f, \Xi_f$ (9) - (hyp.3) + (hyp.5) + (hyp.8) + Confinement (Lemma 7.2)
- $f', \Xi' \sim_{DOM}^{\sigma} f_f', \Xi_f'$ (10) - (hyp.4) + (hyp.6) + (8) + Confinement (Lemma 7.2)
- $f_f, \Xi_f \sim_{DOM}^{\sigma} f_f', \Xi_f'$ **(11)** - (hyp.2) + (9) + (10) + Reflexivity and Symmetry of $\sim_{DOM}^{\sigma}$

In the following suppose that $\sigma_0 \sqcup \sigma_1 \sqcup \Xi(r).\mathsf{value} \sqsubseteq \sigma$ (hyp.8):

- $\sigma_0 \sqcap \sigma_0' \sqsubseteq \sigma \Rightarrow r = r' \wedge \sigma_0 = \sigma_0' \sqsubseteq \sigma$ (12) - (hyp.1) + (1)
- $\sigma_1 \sqcap \sigma_1' \sqsubseteq \sigma \Rightarrow \sigma_1 = \sigma_1' \sqsubseteq \sigma$ (13) - (hyp.1) + (1)
- $v, \sigma_2 \sim_{\sigma} v', \sigma_2'$ (14) - (hyp.1) + (1)
- $\sigma_0 = \sigma_0'$, $\sigma_1 = \sigma_1'$, and $r = r'$ (15) - (hyp.8) + (12) + (13)
- $\Xi(r).\mathsf{value} \sqcap \Xi'(r').\mathsf{value} \sqsubseteq \sigma \Rightarrow \Xi(r).\mathsf{value} = \Xi'(r').\mathsf{value} \sqsubseteq \sigma$ (16) - (hyp.2) + (15)
- $\Xi(r).\mathsf{value} = \Xi'(r').\mathsf{value} \sqsubseteq \sigma$ (17) - (hyp.8) + (16)
- $v, \sigma_f \sim_{\sigma} v', \sigma_f'$ (18) - (14) + (15) + (17)
- $f_f \restriction^{\Xi_f,\sigma} = f_f \restriction^{\Xi_f,\sigma} \setminus \{(r, f(r).\mathsf{value}, \Xi(r).\mathsf{value})\} \cup \{(r, v, \sigma_f)\}$ (19) - (hyp.8) + (6)
- $f_f' \restriction^{\Xi_f',\sigma} = f_f' \restriction^{\Xi_f',\sigma} \setminus \{(r', f'(r').\mathsf{value}, \Xi'(r').\mathsf{value})\} \cup \{(r', v', \sigma_f')\}$ (20) - (hyp.8) + (6)
- $f_f, \Xi_f \sim_{DOM}^{\sigma} f_f', \Xi_f'$ **(21)** - (18) - (20)

In the following suppose that $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \sigma$ (hyp.8):

- $\sigma_0 = \sigma'_0 \sqsubseteq \sigma$, $\sigma_1 = \sigma'_1 \sqsubseteq \sigma$, $\sigma_2 = \sigma_2 \sqsubseteq \sigma$, $r = r'$, and $v = v'$ (22) - (hyp.1) + (hyp.2) + (1)
- $v_f = v'_f$ and $\sigma_f = \sigma'_f$ **(23)** - (2) + (3) + (22)

[Append] $(\mathsf{pg}, \mathsf{pg}_{lab}) = (\mathsf{append}, \mathsf{append}_{lab})$ (hyp.7). We conclude that there are four references r_0, r'_0, r_1, and r'_1 and six security levels σ_0, σ_1, σ_2, σ'_0, σ'_1, and σ_2, and such that:

- $\overrightarrow{v} = r_0 :: \texttt{"appendChild"} :: r_1$, $\overrightarrow{v}' = r'_0 :: \texttt{"appendChild"} :: r'_1$, $\overrightarrow{\sigma} = \sigma_0 :: \sigma_1 :: \sigma_2$, and $\overrightarrow{\sigma}' = \sigma'_0 :: \sigma'_1 :: \sigma'_2$ (1) - (hyp.3) - (hyp.7)
- $v_f = r_2$ and $v'_f = r'_2$ (2) - (hyp.3) + (hyp.4) + (hyp.7)
- $\sigma_f = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \Sigma(r_0).\mathsf{struct} \sqcap \Sigma(r_2).\mathsf{pos}$ (3) - (hyp.5) + (hyp.6) + (hyp.7)
- $\sigma'_f = \sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2 \sqsubseteq \Sigma'(r'_0).\mathsf{struct} \sqcap \Sigma'(r'_2).\mathsf{pos}$ (4) - (hyp.5) + (hyp.6) + (hyp.7)
- The final forest f_f is given by:

$$f_f = f \left[\begin{array}{l} r_0 \mapsto \langle f(r_0).\mathsf{tag}, f(r_0).\mathsf{value}, f(r_0).\mathsf{parent}, f(r_0).\mathsf{children} :: r_2 \rangle, \\ r_2 \mapsto \langle f(r_2).\mathsf{tag}, f(r_2).\mathsf{value}, r_0, f(r_2).\mathsf{children} \rangle \end{array} \right]$$

(5) - (hyp.3) + (hyp.7)

- The final forest f'_f is given by:

$$f'_f = f \left[\begin{array}{l} r'_0 \mapsto \langle f'(r'_0).\mathsf{tag}, f'(r'_0).\mathsf{value}, f'(r'_0).\mathsf{parent}, f'(r'_0).\mathsf{children} :: r'_2 \rangle, \\ r'_2 \mapsto \langle f'(r'_2).\mathsf{tag}, f'(r'_2).\mathsf{value}, r'_0, f'(r'_2).\mathsf{children} \rangle \end{array} \right]$$

(6) - (hyp.4) + (hyp.7)

- $\Xi_f = \Xi$ and $\Xi'_f = \Xi'$ (7) - (hyp.5) + (hyp.6) + (hyp.7)

Suppose $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \not\sqsubseteq \sigma$ (hyp.8). We conclude that:

- $\sigma'_0 \sqcup \sigma'_1 \sqcup \sqsubseteq \sigma'_2 \not\sqsubseteq \sigma$ (8) - (hyp.1) + (hyp.8)
- $f, \Xi \sim^{\sigma}_{DOM} f_f, \Xi_f$ (9) - (hyp.3) + (hyp.5) + (hyp.8) + Confinement (Lemma 7.2 - Append)
- $f', \Xi' \sim^{\sigma}_{DOM} f'_f, \Xi'_f$ (10) - (hyp.4) + (hyp.6) + (8) + Confinement (Lemma 7.2 - Append)
- $f_f, \Xi_f \sim^{\sigma}_{DOM} f'_f, \Xi'_f$ **(11)** - (hyp.2) + (9) + (10) + Reflexivity and Symmetry of $\sim^{\sigma}_{DOM}$

Let $i = |f(r_0).\mathsf{children}|$, $j = |f'(r'_0).\mathsf{children}|$, $\hat{r} = f(r_0).\mathsf{children}(i-1)$, and $\hat{r}' = f'(r'_0).\mathsf{children}(j-1)$. Suppose $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \sigma$ (hyp.8). We conclude that:

- $\sigma_0 = \sigma'_0 \sqsubseteq \sigma$, $r_0 = r'_0$, $\sigma_1 = \sigma'_1$, $\sigma_2 = \sigma'_2$, and $r_2 = r'_2$ (12) - (hyp.1) + (hyp.8) + (1)
- $(\Xi(r_0).\mathsf{struct} = \Xi'(r_0).\mathsf{struct} \sqsubseteq \sigma \wedge i = j) \vee \Xi(r_0).\mathsf{struct} \sqcap \Xi'(r_0).\mathsf{struct} \not\sqsubseteq \sigma$ (13) - (hyp.2) + (12)
- $\Xi(r_0).\mathsf{pos} = \Xi'(r_0).\mathsf{pos} \sqsubseteq \sigma \vee \Xi(r_0).\mathsf{pos} \sqcap \Xi'(r_0).\mathsf{pos} \not\sqsubseteq \sigma$ (14) - (hyp.2) + (12)
- $\Xi(\hat{r}).\mathsf{pos} \sqsubseteq \Xi(r_2).\mathsf{pos}$ and $\Xi'(\hat{r}').\mathsf{pos} \sqsubseteq \Xi'(r'_2).\mathsf{pos}$ (15) - (hyp.2) + (12)
- $(\Xi(r_0).\mathsf{pos} = \Xi'(r_0).\mathsf{pos} \sqsubseteq \sigma \wedge i = j) \vee \Xi(r_0).\mathsf{pos} \sqcap \Xi'(r_0).\mathsf{pos} \not\sqsubseteq \sigma$ (16) - (hyp.2) + (12) + (15)
- $f_f, \Xi_f \sim^{\sigma}_{DOM} f'_f, \Xi'_f$ **(17)** - (hyp.2) + (12) + (13) + (16)

[Remove] Suppose $(\mathsf{pg}, \mathsf{pg}_{lab}) = (\mathsf{remove}, \mathsf{remove}_{lab})$ (hyp.7). We conclude that there are four references r_0, r_2, r'_0, and r'_2 and two integers i and j such that:

- $\overrightarrow{v} = r_0 :: \texttt{"removeChild"} :: r_2$, $\overrightarrow{v}' = r'_0 :: \texttt{"removeChild"} :: r'_2$, $\overrightarrow{\sigma} = \sigma_0 :: \sigma_1 :: \sigma_2$, and $\overrightarrow{\sigma}' = \sigma'_0 :: \sigma'_1 :: \sigma'_2$ (1) - (hyp.3) - (hyp.7)
- $v_f = r_2$ and $v'_f = r'_2$ (2) - (hyp.3) + (hyp.4) + (hyp.7)
- $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \Xi(r_0).\mathsf{struct} \sqcap \Xi(r_2).\mathsf{pos}$ and $\sigma_f = \Xi(r).\mathsf{pos}$ (3) - (hyp.5) + (hyp.7)

- $\sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2 \sqsubseteq \Xi'(r'_0).\mathsf{struct} \sqcap \Xi'(r'_2).\mathsf{pos}$ and $\sigma'_f = \Xi'(r').\mathsf{pos}$ (4) - (hyp.6) + (hyp.7)
- $f(r_0).\mathsf{children}(i) = r_2$ and $f'(r'_0).\mathsf{children}(j) = r'_2$ (5) - (hyp.5) - (hyp.7)
- The final forest f_f is given by:

$$f_f = f \left[\begin{array}{l} r_0 \mapsto \langle f(r_0).\mathsf{tag}, f(r_0).\mathsf{value}, f(r_0).\mathsf{parent}, \mathsf{Shift}_{\mathsf{L}}(f(r_0).\mathsf{children}, i)\rangle, \\ r_2 \mapsto \langle f(r_2).\mathsf{tag}, f(r_2).\mathsf{value}, \mathsf{null}, f(r_2).\mathsf{children}\rangle \end{array} \right]$$

(6) - (hyp.3) + (hyp.7)

- The final forest f'_f is given by:

$$f'_f = f' \left[\begin{array}{l} r'_0 \mapsto \langle f'(r'_0).\mathsf{tag}, f'(r'_0).\mathsf{value}, f'(r'_0).\mathsf{parent}, \mathsf{Shift}_{\mathsf{L}}(f'(r'_0).\mathsf{children}, j)\rangle, \\ r'_2 \mapsto \langle f'(r'_2).\mathsf{tag}, f'(r'_2).\mathsf{value}, \mathsf{null}, f'(r'_2).\mathsf{children}\rangle \end{array} \right]$$

(7) - (hyp.4) + (hyp.7)

In the following suppose that $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \not\sqsubseteq \sigma$ (hyp.8):

- $\sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2 \not\sqsubseteq \sigma$ (7) - (hyp.1) + (hyp.8)
- $f, \Xi \sim^{\sigma}_{DOM} f_f, \Xi_f$ (8) - (hyp.3) + (hyp.5) + (hyp.8) + Confinement (Lemma 7.2 - Remove)
- $f', \Xi' \sim^{\sigma}_{DOM} f'_f, \Xi'_f$ (9) - (hyp.4) + (hyp.6) + (7) + Confinement (Lemma 7.2 - Remove)
- $f_f, \Xi_f \sim^{\sigma}_{DOM} f'_f, \Xi'_f$ **(11)** - (hyp.2) + (8) + (9) + Reflexivity and Symmetry of $\sim^{\sigma}_{DOM}$

In the following suppose that $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \sigma$ (hyp.8):

- $r_0 = r'_0$, $\sigma_0 = \sigma'_0$, $\sigma_1 = \sigma'_1$, $\sigma_2 = \sigma'_2$, and $r_2 = r'_2$ (12) - (hyp.1) + (hyp.8)
- $(\Xi(r_0).\mathsf{struct} = \Xi'(r_0).\mathsf{struct} \sqsubseteq \sigma \wedge |f(r_0).\mathsf{children}| = |f'(r_0).\mathsf{children}|) \vee$
  $\vee\ (\Xi(r_0).\mathsf{struct} \sqcap \Xi'(r_0).\mathsf{struct} \not\sqsubseteq \sigma)$ (13) - (hyp.2) + (12)
- $(\Xi(r_2).\mathsf{pos} = \Xi'(r'_2).\mathsf{pos} \sqsubseteq \sigma \wedge i = j \wedge f(r_2).\mathsf{parent} = f'(r_2).\mathsf{parent}) \vee$
  $\big(\forall_{k \geq i} \Xi(f(r_0).\mathsf{children}(k)).\mathsf{pos} \not\sqsubseteq \sigma \wedge \forall_{k \geq j} \Xi'(f'(r_0).\mathsf{children}(k)).\mathsf{pos} \not\sqsubseteq \sigma \wedge$
  $\wedge \Xi(r_2).\mathsf{parent} \sqcap \Xi'(r_2).\mathsf{parent} \not\sqsubseteq \sigma\big)$ (14) - (hyp.2) + (12)
- $f_f, \Xi_f \sim^{\sigma}_{DOM} f'_f, \Xi'_f$ **(15)** - (hyp.2) + (12)-(14)

□

D.2 Proving Low-Equality Strengthening

Definitions D.1 and D.2 strengthen the low-equality for sequences introduced in the previous section. Definition D.1 requires that the two sequences coincide in their low prefix, while Definition D.2 require that they entirely coincide.

Definition D.1 (Asymmetric Low-Equality for Sequences)**.** *Two lists of values $\overrightarrow{v}$ and $\overrightarrow{v}'$ respectively labelled by two lists of security levels $\overrightarrow{\sigma}$ and $\overrightarrow{\sigma}'$ are said to be asymmetrically low-equal w.r.t. a security level σ, written $\overrightarrow{v}, \overrightarrow{\sigma} \simeq_{\sigma} \overrightarrow{v}', \overrightarrow{\sigma}'$ if the following hold there is an integer i such that:* **(1)** $\forall_{0 \leq j < i}\ \overrightarrow{\sigma}(i) = \overrightarrow{\sigma}'(i) \sqsubseteq \sigma \ \wedge\ \overrightarrow{v}(i) = \overrightarrow{v}'(i)$, **(2)** $\forall_{i \leq j < |\overrightarrow{v}|}\ \overrightarrow{\sigma}(i) \not\sqsubseteq \sigma$, *and* **(3)** $\forall_{i \leq j < |\overrightarrow{v}'|}\ \overrightarrow{\sigma}'(i) \not\sqsubseteq \sigma$. *Furthermore, for all security levels σ, it holds that $\varepsilon, \varepsilon \simeq_{\sigma} \varepsilon, \varepsilon$.*

Definition D.2 (Strong Low-Equality for Sequences)**.** *Two lists of values $\overrightarrow{v}$ and $\overrightarrow{v}'$ respectively labelled by two lists of security levels $\overrightarrow{\sigma}$ and $\overrightarrow{\sigma}'$ are said to be strongly low-equal w.r.t. a security level σ, written $\overrightarrow{v}, \overrightarrow{\sigma} \approx_{\sigma} \overrightarrow{v}', \overrightarrow{\sigma}'$ if:* **(1)** $|\overrightarrow{v}| = |\overrightarrow{v}'|$ *and* **(2)** $\forall_{0 \leq i < |\overrightarrow{v}|}\ \overrightarrow{\sigma}(i) = \overrightarrow{\sigma}'(i) \sqsubseteq \sigma \ \wedge\ \overrightarrow{v}(i) = \overrightarrow{v}'(i)$.

In the following, given a function g mapping references to security levels and a sequence of reference $\overrightarrow{r}$, we use $g(\overrightarrow{r})$ to denote the sequence $\overrightarrow{\sigma}$ obtained by applying g to every element of $\overrightarrow{r}$. Formally, $\overrightarrow{\sigma}$ is such that: $|\overrightarrow{r}| = |\overrightarrow{\sigma}|$ and for all $0 \leq i < |\overrightarrow{r}|$: $\overrightarrow{\sigma}(i) = g(\overrightarrow{r}(i))$. We use $\Xi.\mathsf{pos}$ to denote the function that maps each node reference to the corresponding position level. Formally, $\Xi.\mathsf{pos}(r) = \Xi(r).\mathsf{pos}$. Moreover, given a list of security level $\overrightarrow{\sigma}$ and a security level σ, we use $\sigma \sqsubseteq \overrightarrow{\sigma}$ as an abbreviation for $\sigma \sqsubseteq \sqcap\{\overrightarrow{\sigma}(i) \mid 0 \leq i < |\overrightarrow{\sigma}|\}$ and $\overrightarrow{\sigma} \sqsubseteq \sigma$ as an abbreviation for $\sqcup\{\overrightarrow{\sigma}(i) \mid 0 \leq i < |\overrightarrow{\sigma}|\} \sqsubseteq \sigma$.

Lemma D.1 states that, given a well-labelled tree for live collections, if one searches for the nodes with a given tag m starting from the root of that tree, one obtains a sequence of nodes whose position levels are monotonically increasing.

Lemma D.1 (Monotonocity of the search relation)**.** *Given a forest f labelled by Ξ, a node reference r, a function ϕ_ℓ, and a tag name m such that $\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\ell \rightsquigarrow \phi'_\ell$ and $f \vdash r \rightsquigarrow_m \overrightarrow{r}$, for a given function ϕ'_ℓ and list of references $\overrightarrow{r}$, it holds that:* **(1)** *$\Xi.\mathsf{pos}(\overrightarrow{r})$ is monotonically increasing and* **(2)** *$\phi_\ell(m) \sqsubseteq \Xi.\mathsf{pos}(\overrightarrow{r}) \sqsubseteq \phi'_\ell(m) \sqsubseteq \sigma_m$.*

Proof: We begin by restating the hypotheses of the lemma:

- $\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\ell \rightsquigarrow \phi'_\ell$ (hyp.1)
- $f \vdash r \rightsquigarrow_m \overrightarrow{r}$ (hyp.2)

The proof proceeds by induction on the structure of the derivation of $f \vdash r \rightsquigarrow_m \overrightarrow{r}$. There are two base cases to consider [Node Not Found - Leaf Node] and [Node Found - Leaf Node]. The inductive cases are [Node Not Found - Non-Leaf Node] and [Node Found - Non-Leaf Node]. Since the inductive case are analogous, we only consider the case [Node Found - Non-Leaf Node], which is the most complex.

[Node Not Found - Leaf Node] In this case: $|f(r).\mathsf{children}| = 0$ and $f(r).\mathsf{tag} \neq m$ (hyp.3). We conclude that:

- $\phi'_\ell = \phi_\ell$ and $\overrightarrow{r} = \varepsilon$ **(1)** - (hyp.1)-(hyp.3)

[Node Found - Leaf Node] In this case: $|f(r).\mathsf{children}| = 0$, $f(r).\mathsf{tag} = m$, (hyp.3). Letting $\sigma = \Xi(r).\mathsf{pos}$, we conclude that:

- $\phi_\ell(m) \sqsubseteq \sigma \sqsubseteq \sigma_m$, $\phi'_\ell = \phi_\ell\,[m \mapsto \sigma]$, and $\overrightarrow{r} = r :: \varepsilon$ (1) - (hyp.1)-(hyp.3)
- $\phi_\ell(m) \sqsubseteq \Xi(\overrightarrow{r}(0)).\mathsf{pos} = \phi'_\ell(m) \sqsubseteq \sigma_m$ **(2)** - (1)

[Node Found - Non-Leaf Node] In this case: $|f(r).\mathsf{children}| = n$, $n \neq 0$, $f(r).\mathsf{tag} = m$, (hyp.3). Letting $\sigma = \Xi(r).\mathsf{pos}$ and $\overrightarrow{r}' = f(r).\mathsf{children}$, we conclude that:

- $\overrightarrow{r} = r :: \overrightarrow{r}_0 :: \cdots :: \overrightarrow{r}_n$, where $f \vdash \overrightarrow{r}(i) \rightsquigarrow_m \overrightarrow{r}_i$ for $0 \leq i < n$ (1) - (hyp.1) + (hyp.3)
- $\phi_\ell(m) \sqsubseteq \sigma \sqsubseteq \sigma_m$ (2) - (hyp.2) + (hyp.3)
- $\phi_\ell(m) \sqsubseteq \Xi(r).\mathsf{pos} = \phi^0_\ell(m) \sqsubseteq \sigma_m$, where $\phi^0_\ell = \phi_\ell\,[m \mapsto \sigma]$ (3) - (hyp.2) + (hyp.3) + (2)
- $\forall_{0 \leq i < n}\ \mathcal{S}ec_{f,\Xi} \vdash^{f(r).\mathsf{children}(i)} \phi^i_\ell \rightsquigarrow \phi^{i+1}_\ell$ and $\phi'_\ell = \phi^n_\ell$ (4) - (hyp.2) + (hyp.3)
- For all $0 \leq i < n$, $\Xi.\mathsf{pos}(\overrightarrow{r}_i)$ is monotonically increasing and:

  $$\phi^i_\ell(m) \sqsubseteq \Xi.\mathsf{pos}(\overrightarrow{r}_i) \sqsubseteq \phi^{i+1}_\ell(m) \sqsubseteq \sigma_m$$

  (5) - (1) + (4) + **ih**
- The list $\overrightarrow{r}$ is monotonically increasing and $\phi_\ell(m) \sqsubseteq \Xi.\mathsf{pos}(\overrightarrow{r}) \sqsubseteq \phi'_\ell(m) \sqsubseteq \sigma_m$

  (6) - (3) + (5)

$$\textsc{Leaf Node}\quad \frac{\begin{array}{cc} f(r).\mathsf{tag} = m & |f(r).\mathsf{children}| = 0 \\ \sigma = \Xi(r).\mathsf{pos} & \phi_\natural(m) \sqsubseteq \sigma \sqsubseteq \sigma_m \\ \phi'_\natural = \phi_\natural\,[m \mapsto \sigma] & \varphi'_\natural = \varphi_\natural\,[(m,\sigma) \mapsto r] \end{array}}{\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural, \varphi_\natural \rightsquigarrow \phi'_\natural, \varphi'_\natural}$$

$$\textsc{Non-Leaf Node}\quad \frac{\begin{array}{cc} f(r).\mathsf{tag} = m & \sigma = \Xi(r).\mathsf{pos} \\ \phi_\natural(m) \sqsubseteq \sigma \sqsubseteq \sigma_m & |f(r).\mathsf{children}| = n > 0 \\ \phi^0_\natural = \phi_\natural\,[m \mapsto \sigma] & \varphi^0_\natural = \varphi_\natural\,[(m,\sigma) \mapsto r] \\ \multicolumn{2}{c}{\forall_{0\le i<n}\ \Xi(r).\mathsf{pos} \sqsubseteq \Xi(f(r).\mathsf{children}(i)).\mathsf{pos}} \\ \multicolumn{2}{c}{\forall_{0\le i<n}\ \mathcal{S}ec_{f,\Xi} \vdash^{f(r).\mathsf{children}(i)} \phi^i_\natural, \varphi^i_\natural \rightsquigarrow \phi^{i+1}_\natural, \varphi^{i+1}_\natural} \end{array}}{\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural, \varphi_\natural \rightsquigarrow \phi^n_\natural, \varphi^n_\natural}$$

Figure D.1: Well-labelling Predicate for Live Primitives

□

In order to be able to prove Theorem 7.2, we need to reason about low-equal trees that are well-labelled for live collections. Therefore, one must modify the *well-labelling* predicate for it to compute the additional information required for the proofs. Concretely, the new version of the predicate (which is presented in Figure D.1) computes for each tree a function $\varphi_\natural$, called *live record*, that maps every pair (m, σ), consisting of a tag name and a security level, to the last node in that tree (in document order) with tag m whose position level is $\sqsubseteq \sigma$. Given a live record $\varphi_\natural$, we define its *low-projection* at level σ, written $\varphi_\natural \restriction^\sigma$, as the live record $\varphi'_\natural$ given by:

$$\varphi'_\natural(m,\sigma') = \begin{cases} \varphi_\natural(m,\sigma') & \text{if } \sigma' \sqsubseteq \sigma \\ \textit{undefined} & \text{otherwise} \end{cases}$$

If a tree is well-labelled for a live collections and the position level of its root node is not observable, then applying the definition, we conclude that the position levels of all the nodes in that tree are not observable. This means that, if the the position level of a node n is not observable at level σ, the low-projection at σ of the live record computed by searching the tree rooted at n coincides with the low-projection at σ of the initial live record. This fact is formally established in Lemma D.2. Furthermore, if two trees are well-labelled for live collections and low-equal at a given security level σ (according to the first low-equality for trees - $\sim^\sigma_{DOM}$), then the low-projections of their respective live records at level σ coincide. This is established in Lemma D.3.

Lemma D.2 (Highly-Positioned Tree)**.** *Given a forest f labelled by Ξ, a node reference r, two functions $\phi_\natural$ and $\varphi_\natural$, and a tag name m such that $\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural, \varphi_\natural \rightsquigarrow \phi'_\natural, \varphi'_\natural$ and $\Xi(r).\mathsf{pos} \not\sqsubseteq \sigma$, for some functions $\phi'_\natural$ and $\varphi'_\natural$, it holds that: $\varphi_\natural \restriction^\sigma = \varphi'_\natural \restriction^\sigma$.*

Proof: Given that:

- $\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural, \varphi_\natural \rightsquigarrow \phi'_\natural, \varphi'_\natural$ (hyp.1)
- $\Xi(r).\mathsf{pos} \not\sqsubseteq \sigma$ (hyp.2)

we have to prove that $\varphi_\natural \restriction^\sigma = \varphi'_\natural \restriction^\sigma$. We proceed by induction on the derivation of (hyp.1). The base case is [Leaf Node] and the inductive case is [Non-Leaf Node].

[Leaf Node] In this case: $|f(r).\mathsf{children}| = 0$ (hyp.3). Letting $m = f(r).\mathsf{tag}$ and $\sigma' = \Xi(r).\mathsf{pos}$, we conclude that:

- $\varphi'_\natural = \varphi_\natural\,[(m,\sigma') \mapsto r]$ (1) - (hyp.1) + (hyp.3)
- $\varphi'_\natural \restriction^\sigma = \varphi_\natural \restriction^\sigma$ **(2)** - (hyp.2) + (1)

[Non-Leaf Node] In this case: $|f(r).\mathsf{children}| = n$ for $n > 0$ (hyp.3). Letting $m = f(r).\mathsf{tag}$, $\sigma' = \Xi(r).\mathsf{pos}$, $\phi_\natural^0 = \phi_\natural\,[m \mapsto \sigma]$, and $\varphi_\natural^0 = \varphi_\natural\,[(m,\sigma) \mapsto r]$, we conclude that:

- $\forall_{0\leq i<n}\ \Xi(r).\mathsf{pos} \sqsubseteq \Xi(f(r).\mathsf{children}(i)).\mathsf{pos}$ (1) - (hyp.1) + (hyp.3)
- $\forall_{0\leq i<n}\ \mathcal{S}ec_{f,\Xi} \vdash^{f(r).\mathsf{children}(i)} \phi_\natural^i, \varphi_\natural^i \rightsquigarrow \phi_\natural^{i+1}, \varphi_\natural^{i+1}$ and $\varphi'_\natural = \varphi_\natural^n$ (2) - (hyp.1) + (hyp.3)
- $\varphi_\natural^0 \restriction^\sigma = \varphi_\natural \restriction^\sigma$ (3) - *definition*
- $\forall_{0\leq i<n}\ \Xi(f(r).\mathsf{children}(i)).\mathsf{pos} \not\sqsubseteq \sigma$ (4) - (hyp.2) + (1)
- $\forall_{0\leq i<n}\ \varphi_\natural^i \restriction^\sigma = \varphi_\natural^{i+1} \restriction^\sigma$ (5) - (2) + (4) + **ih**
- $\varphi_\natural^0 \restriction^\sigma = \varphi_\natural^n \restriction^\sigma = \varphi'_\natural \restriction^\sigma$ (6) - (2) + (5)
- $\varphi_\natural \restriction^\sigma = \varphi'_\natural \restriction^\sigma$ **(7)** - (3) + (6)

□

Lemma D.3 (Live Records of Well-labelled Low-Equal Trees)**.** *Given two forests f and $\hat{f}$ respectively well-labelled by Ξ and $\hat{\Xi}$, two live functions $\phi_\natural$ and $\hat{\phi}_\natural$, two live records $\varphi_\natural$ and $\hat{\varphi}_\natural$, a node reference r, and a security level σ such that: $\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural, \varphi_\natural \rightsquigarrow \phi'_\natural, \varphi'_\natural$, $\mathcal{S}ec_{\hat{f},\hat{\Xi}} \vdash^r \hat{\phi}_\natural, \hat{\varphi}_\natural \rightsquigarrow \hat{\phi}'_\natural, \hat{\varphi}'_\natural$, $f, \Xi \sim^\sigma_{DOM} \hat{f}, \hat{\Xi}$, and $\varphi_\natural \restriction^\sigma = \hat{\varphi}_\natural \restriction^\sigma$, it holds that: $\varphi'_\natural \restriction^\sigma = \hat{\varphi}'_\natural \restriction^\sigma$.*

Proof: Given that:

- $\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural, \varphi_\natural \rightsquigarrow \phi'_\natural, \varphi'_\natural$ (hyp.1)
- $\mathcal{S}ec_{\hat{f},\hat{\Xi}} \vdash^r \hat{\phi}_\natural, \hat{\varphi}_\natural \rightsquigarrow \hat{\phi}'_\natural, \hat{\varphi}'_\natural$ (hyp.2)
- $f, \Xi \sim^\sigma_{DOM} \hat{f}, \hat{\Xi}$ (hyp.3)
- $\varphi_\natural \restriction^\sigma = \hat{\varphi}_\natural \restriction^\sigma$ (hyp.4)

we have to prove that $\varphi'_\natural \restriction^\sigma = \hat{\varphi}'_\natural \restriction^\sigma$. Suppose that $\Xi(r).\mathsf{pos} \not\sqsubseteq \sigma$ (hyp.5). We conclude that:

- $\varphi'_\natural \restriction^\sigma = \varphi_\natural \restriction^\sigma$ (1) - (hyp.1) + (hyp.5) + *High Positioned Tree*
- $\hat{\Xi}(r).\mathsf{pos} \not\sqsubseteq \sigma$ (2) - (hyp.3) + (hyp.5)
- $\hat{\varphi}'_\natural \restriction^\sigma = \hat{\varphi}_\natural \restriction^\sigma$ (3) - (hyp.2) + (2) + *High Positioned Tree*
- $\varphi'_\natural \restriction^\sigma = \hat{\varphi}'_\natural \restriction^\sigma$ **(4)** - (hyp.4) + (1) + (3)

In the rest of the proof we suppose that $\Xi(r).\mathsf{pos} = \hat{\Xi}(r).\mathsf{pos} \sqsubseteq \sigma$ (hyp.5) and we proceed by induction on the derivation of (hyp.1). The base case is [Leaf Node] and the inductive case is [Non-Leaf Node].

[Leaf Node] In this case: $|f(r).\mathsf{children}| = 0$ (hyp.6). Letting $m = f(r).\mathsf{tag}$ and $\sigma' = \Xi(r).\mathsf{pos} = \hat{\Xi}(r).\mathsf{pos}$, $\hat{m} = \hat{f}(r).\mathsf{tag}$, and $\hat{\varphi}_\natural^0 = \hat{\varphi}_\natural\,[(\hat{m},\sigma') \mapsto r]$, we conclude that:

- $\varphi'_\natural = \varphi_\natural\,[(m,\sigma') \mapsto r]$ (1) - (hyp.1) + (hyp.6)
- $\Xi(r).\mathsf{node} = \hat{\Xi}(r).\mathsf{node} \sqsubseteq \sigma$ (2) - (hyp.3) + (hyp.5)
- $m = \hat{m}$ (3) - (hyp.3) + (2)
- $\varphi'_\natural \restriction^\sigma = \hat{\varphi}_\natural^0 \restriction^\sigma$ (4) - (hyp.4) + (hyp.5) + (1) + (3)

If $|\hat{f}(r).\mathsf{children}| = 0$, then $\hat{\varphi}'_\natural = \hat{\varphi}_\natural^0$ and the result follows immediately by (4). Hence, suppose that: $|\hat{f}(r).\mathsf{children}| = n > 0$ (hyp.7). We conclude that:

- $\forall_{0\leq i<n}\ \mathcal{S}ec_{\hat{f},\hat{\Xi}} \vdash^{\hat{f}(r).\mathsf{children}(i)} \hat{\phi}_\natural^i, \hat{\varphi}_\natural^i \rightsquigarrow \hat{\phi}_\natural^{i+1}, \hat{\varphi}_\natural^{i+1}$ and $\hat{\varphi}'_\natural = \varphi_\natural^n$ (5) - (hyp.2) + (hyp.7)

- $\hat{\Xi}(r).\mathsf{struct} \not\sqsubseteq \sigma$ (6) - (hyp.3) + (hyp.6) + (hyp.7)
- $\forall_{0\leq i<n}\ \hat{\Xi}(\hat{f}(r).\mathsf{children}(i)).\mathsf{pos} \not\sqsubseteq \sigma$ (7) - (hyp.2) + (6)
- $\forall_{0\leq i<n}\ \hat{\varphi}^i_\ell\ \restriction^\sigma = \hat{\varphi}^{i+1}_\ell\ \restriction^\sigma$ (8) - (5) + (7) + *High-Positioned Tree*
- $\hat{\varphi}^0_\ell\ \restriction^\sigma = \hat{\varphi}^n_\ell\ \restriction^\sigma = \hat{\varphi}'_\ell\ \restriction^\sigma$ (9) - (5) + (8)
- $\varphi'_\ell\ \restriction^\sigma = \hat{\varphi}'_\ell\ \restriction^\sigma$ **(10)** - (4) + (9)

[Non-Leaf Node] In this case: $|f(r).\mathsf{children}| = n > 0$ (hyp.6). Since the case in which $|\hat{f}(r).\mathsf{children}| = 0$ is symmetric to the previous case. We shall assume that: $|\hat{f}(r).\mathsf{children}| = \hat{n} > 0$ (hyp.7). Letting $m = f(r).\mathsf{tag}$ and $\sigma' = \Xi(r).\mathsf{pos} = \hat{\Xi}(r).\mathsf{pos}$, $\hat{m} = \hat{f}(r).\mathsf{tag}$, $\varphi^0_\ell = \varphi_\ell\,[(m,\sigma') \mapsto r]$, $\hat{\varphi}^0_\ell = \hat{\varphi}_\ell\,[(\hat{m},\sigma') \mapsto r]$, we conclude that:

- $\Xi(r).\mathsf{node} = \hat{\Xi}(r).\mathsf{node} \sqsubseteq \sigma$ (1) - (hyp.3) + (hyp.5)
- $m = \hat{m}$ (2) - (hyp.3) + (1)
- $\forall_{0\leq i<n}\ \mathcal{S}ec_{f,\Xi} \vdash^{f(r).\mathsf{children}(i)} \phi^i_\ell, \varphi^i_\ell \rightsquigarrow \phi^{i+1}_\ell, \varphi^{i+1}_\ell$ and $\varphi'_\ell = \varphi^n_\ell$ (3) - (hyp.1) + (hyp.6)
- $\forall_{0\leq i<\hat{n}}\ \mathcal{S}ec_{\hat{f},\hat{\Xi}} \vdash^{\hat{f}(r).\mathsf{children}(i)} \hat{\phi}^i_\ell, \hat{\varphi}^i_\ell \rightsquigarrow \hat{\phi}^{i+1}_\ell, \hat{\varphi}^{i+1}_\ell$ and $\hat{\varphi}'_\ell = \varphi^{\hat{n}}_\ell$ (4) - (hyp.2) + (hyp.7)
- $\varphi^0_\ell\ \restriction^\sigma = \hat{\varphi}^0_\ell\ \restriction^\sigma$ (5) - (hyp.4) + (1) + (2)

Since the position levels of the children of $f(r)$ and $\hat{f}(r)$ are in increasing order, we conclude from (hyp.3) that there is an unique integer j such that:

- $\forall_{0\leq i<j}\ \Xi(f(r).\mathsf{children}(i)).\mathsf{pos} \sqsubseteq \sigma$ (6) - (hyp.3)
- $\forall_{j\leq i<|f(r).\mathsf{children}|}\ \Xi(f(r).\mathsf{children}(i)).\mathsf{pos} \not\sqsubseteq \sigma$ (7) - (hyp.3)
- $\forall_{0\leq i<j}\ \hat{\Xi}(\hat{f}(r).\mathsf{children}(i)).\mathsf{pos} \sqsubseteq \sigma$ (8) - (hyp.3)
- $\forall_{j\leq i<|\hat{f}(r).\mathsf{children}|}\ \hat{\Xi}(\hat{f}(r).\mathsf{children}(i)).\mathsf{pos} \not\sqsubseteq \sigma$ (9) - (hyp.3)
- $\varphi^j_\ell\ \restriction^\sigma = \varphi^n_\ell\ \restriction^\sigma = \varphi'_\ell\ \restriction^\sigma$ (10) - (3) + (7)
- $\hat{\varphi}^j_\ell\ \restriction^\sigma = \hat{\varphi}^n_\ell\ \restriction^\sigma = \hat{\varphi}'_\ell\ \restriction^\sigma$ (11) - (4) + (9)
- $\forall_{0\leq i<j}\ f(r).\mathsf{children}(i) = \hat{f}(r).\mathsf{children}(i)$ (12) - (hyp.3)

We now prove by induction on j that $\varphi^j_\ell\ \restriction^\sigma = \hat{\varphi}^j_\ell\ \restriction^\sigma$. If $j = 0$, then the result immediately holds by (5). Suppose that $j = k + 1$:

- $\varphi^k_\ell\ \restriction^\sigma = \hat{\varphi}^k_\ell\ \restriction^\sigma$ (13) - **inner ih**
- $\varphi^{k+1}_\ell\ \restriction^\sigma = \hat{\varphi}^{k+1}_\ell\ \restriction^\sigma$ (14) - (hyp.3) + (3) + (4) + (12) + (13) + **outer ih**

□

The following two lemmas state two simple invariance properties that computed live records observe. Suppose that one searches the nodes with tag m of a given subtree of a well-labelled tree. And, before starting the search, the current live record (φ_ℓ) already contains a node with tag m and position level σ ($(m,\sigma) \in dom(\varphi_\ell)$). Since, the whole tree is assumed to be well-labelled, it follows that all the nodes with tag m in that subtree must have position levels greater than or equal to σ. This means that for all levels σ' such that $\sigma \not\sqsubseteq \sigma'$, the final live record (φ'_ℓ) coincides with the initial ($\varphi'_\ell(m,\sigma) = \varphi_\ell(m,\sigma)$). This is established in Lemma D.4.

Lemma D.5 establishes a dual property of the one just explained above. It says that whenever the final live record coincides with the initial live record for a given tag name m and security level σ, it is because one of the following two propositions holds:

- No nodes with tag m and security level σ were found when traversing the tree;
- Every node with tag m found when traversing the tree had a security level strictly greater than σ.

Lemma D.4 (Live Record Invariance - 1)**.** *Given a forest f labelled by Ξ, a live function $\phi_\natural$, a live record $\varphi_\natural$, a node reference r, a tag name m, and two security levels σ and σ' such that: $\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural, \varphi_\natural \rightsquigarrow \phi'_\natural, \varphi'_\natural$, $(m, \sigma) \in dom(\varphi_\natural)$, and $\sigma \not\sqsubseteq \sigma'$, it holds that: $\varphi_\natural(m, \sigma') = \varphi'_\natural(m, \sigma')$.*

Proof: If $(m, \sigma) \in dom(\varphi_\natural)$, then in order for the subtree rooted in r to be well-labelled by Ξ (which it is), all the nodes with tag m that it includes must have a position level higher than or equal to σ. Therefore, we conclude that it does not include any node with tag m whose position level is $\not\sqsupseteq \sigma$, from which the result follows. □

Lemma D.5 (Live Record Invariance - 2)**.** *Given a forest f labelled by Ξ, a live function $\phi_\natural$, a live record $\varphi_\natural$, a node reference r, a tag name m, and a security level σ such that: $\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural, \varphi_\natural \rightsquigarrow \phi'_\natural, \varphi'_\natural$, $f \vdash r \rightsquigarrow_m \overrightarrow{r}$, and $\varphi_\natural(m, \sigma) = \varphi'_\natural(m, \sigma)$, it holds that $\forall_{0 \leq i < |\overrightarrow{r}|}\ \Xi(\overrightarrow{r}(i)).\mathsf{pos} \not\sqsubseteq \sigma$.*

Proof: Given that:

- $\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural, \varphi_\natural \rightsquigarrow \phi'_\natural, \varphi'_\natural$ (hyp.1)
- $f \vdash r \rightsquigarrow_m \overrightarrow{r}$ (hyp.2)
- $\varphi_\natural(m, \sigma) = \varphi'_\natural(m, \sigma)$ (hyp.3)

it holds that $\forall_{0 \leq i < |\overrightarrow{r}|}\ \Xi(\overrightarrow{r}(i)).\mathsf{pos} \not\sqsubseteq \sigma$. The proof proceeds by induction on the structure of the derivation of $f \vdash r \rightsquigarrow_m \overrightarrow{r}$. There are two base cases to consider [Node Not Found - Leaf Node] and [Node Found - Leaf Node]. The inductive cases are [Node Not Found - Non-Leaf Node] and [Node Found - Non-Leaf Node]. Since the inductive case are analogous, we only consider the case [Node Found - Non-Leaf Node], which is the most complex.

[Node Not Found - Leaf Node] In this case: $|f(r).\mathsf{children}| = 0$. Hence the result holds vacuously.

[Node Found - Leaf Node] In this case: $|f(r).\mathsf{children}| = 0$ and $f(r).\mathsf{tag} = m$ (hyp.4). Letting $\sigma' = \Xi(r).\mathsf{pos}$, we conclude that:

- $\varphi'_\natural = \varphi_\natural\ [(m, \sigma') \mapsto r]$ (1) - (hyp.1) + (hyp.4)
- $\overrightarrow{r} = \sigma' :: \varepsilon$ (2) - (hyp.2) + (hyp.4)
- $\sigma' \neq \sigma$ (3) - (hyp.3) + (1)
- $\sigma' \not\sqsubseteq \sigma$ **(4)**
  Suppose that: $\sigma' \sqsubseteq \sigma$ (hyp.4). We conclude that:
  - $\sigma' \sqsubset \sigma$ (4.1) - (hyp.4) + (3)
  - $\sigma \not\sqsubseteq \sigma'$ (4.2) - (4.1)
  - $\varphi_\natural(m, \sigma') = \varphi'_\natural(m, \sigma')$ (4.3) - (4.2) + *Live Record Invariance - 1*
  - *Contradiction* (4.4) - (1) + (4.3)

[Node Found - Non-Leaf Node] In this case: $|f(r).\mathsf{children}| = n > 0$, and $f(r).\mathsf{tag} = m$ (hyp.4). Letting $\sigma' = \Xi(r).\mathsf{pos}$ and $\overrightarrow{r}' = f(r).\mathsf{children}$, we conclude that:

- $\overrightarrow{r}' = r :: \overrightarrow{r}_0 :: \cdots :: \overrightarrow{r}_{n-1}$, where: $f \vdash \overrightarrow{r}(i) \rightsquigarrow_m \overrightarrow{r}_i$ for $0 \leq i < n$ (1) - (hyp.2) + (hyp.4)
- $\varphi^0_\natural = \varphi_\natural\ [(m, \sigma') \mapsto r]$ (2) - (hyp.1) + (hyp.4)

- $\sigma' \neq \sigma$ (3) - (hyp.3) + (2)
- $\sigma' \not\sqsubseteq \sigma$ (4) - (hyp.4) + (2) + (3)
- $\forall_{0 \leq i < n}\ \mathcal{S}ec_{f,\Xi} \vdash^{f(r).\mathsf{children}(i)} \phi_\natural^i, \varphi_\natural^i \rightsquigarrow \phi_\natural^{i+1}, \varphi_\natural^{i+1}$ and $\varphi'_\natural = \varphi_\natural^n$ (5) - (hyp.1) + (hyp.4)
- $\forall_{0 \leq i < n}\ \varphi_\natural^i(m, \sigma) = \varphi_\natural^{i+1}(m, \sigma)$ (6) - (hyp.1) + (hyp.3) + (hyp.4)
- $\forall_{0 \leq i < n}\ \forall_{0 \leq j < |\overrightarrow{r}_i|}\ \Xi(\overrightarrow{r}_i(j)).\mathsf{pos} \not\sqsubseteq \sigma$ (7) - (1) + (5) + (6) + **ih**
- $\forall_{0 \leq i < |\overrightarrow{r}|}\ \Xi(\overrightarrow{r}(i)).\mathsf{pos} \not\sqsubseteq \sigma$ **(8)** - (1) + (7)

□

Finally, Lemma D.6 states the main property required for the proof of Theorem 7.2. In a nutshell, it says that the sequences of nodes obtained when searching for the nodes with a given tag in two well-labelled and low-equal trees at a given level σ ($\sim^{\sigma}_{DOM}$) are asymmetrically low-equal when labelled with the respective position levels.

Lemma D.6 (Low-Equal DOM Searches)**.** *Given two forests f and $\hat{f}$ respectively labelled by Ξ and $\hat{\Xi}$, two live functions $\phi_\natural$ and $\hat{\phi}_\natural$, two live records $\varphi_\natural$ and $\hat{\varphi}_\natural$, a node reference r, and a security level σ such that: $\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural, \varphi_\natural \rightsquigarrow \phi'_\natural, \varphi'_\natural$, $\mathcal{S}ec_{\hat{f},\hat{\Xi}} \vdash^r \hat{\phi}_\natural, \hat{\varphi}_\natural \rightsquigarrow \hat{\phi}'_\natural, \hat{\varphi}'_\natural$, $f \vdash r \rightsquigarrow_m \overrightarrow{r}$, $\hat{f} \vdash r \rightsquigarrow_m \hat{\overrightarrow{r}}$, $f, \Xi \sim^{\sigma}_{DOM} \hat{f}, \hat{\Xi}$, and $\varphi_\natural \restriction^{\sigma} = \hat{\varphi}_\natural \restriction^{\sigma}$,* **it holds that:** *$\overrightarrow{r}, \Xi.\mathsf{pos}(\overrightarrow{r}) \simeq_\sigma \hat{\overrightarrow{r}}, \hat{\Xi}.\mathsf{pos}(\hat{\overrightarrow{r}})$.*

Proof: Given that:

- $\mathcal{S}ec_{f,\Xi} \vdash^r \phi_\natural, \varphi_\natural \rightsquigarrow \phi'_\natural, \varphi'_\natural$ (hyp.1),
- $\mathcal{S}ec_{\hat{f},\hat{\Xi}} \vdash^r \hat{\phi}_\natural, \hat{\varphi}_\natural \rightsquigarrow \hat{\phi}'_\natural, \hat{\varphi}'_\natural$ (hyp.2),
- $f \vdash r \rightsquigarrow_m \overrightarrow{r}$ (hyp.3)
- $\hat{f} \vdash r \rightsquigarrow_m \hat{\overrightarrow{r}}$ (hyp.4)
- $f, \Xi \sim^{\sigma}_{DOM} \hat{f}, \hat{\Xi}$ (hyp.5)
- $\varphi_\natural \restriction^{\sigma} = \hat{\varphi}_\natural \restriction^{\sigma}$ (hyp.6)

It holds that: $\overrightarrow{r}, \Xi.\mathsf{pos}(\overrightarrow{r}) \simeq_\sigma \hat{\overrightarrow{r}}, \hat{\Xi}.\mathsf{pos}(\hat{\overrightarrow{r}})$. Suppose that $\Xi(r).\mathsf{pos} \not\sqsubseteq \sigma$ (hyp.7), we conclude that:

- $\hat{\Xi}(r).\mathsf{pos} \not\sqsubseteq \sigma$ (1) - (hyp.5) + (hyp.7)
- $\forall_{0 \leq i < |\overrightarrow{r}|}\ \Xi(\overrightarrow{r}(i)).\mathsf{pos} \not\sqsubseteq \sigma$ (2) - (hyp.1) + (hyp.3) + (hyp.7) + *Monotonocity of the Search Relation*
- $\forall_{0 \leq i < |\hat{\overrightarrow{r}}|}\ \hat{\Xi}(\hat{\overrightarrow{r}}(i)).\mathsf{pos} \not\sqsubseteq \sigma$ (3) - (hyp.2) + (hyp.4) + (1) + *Monotonocity of the Search Relation*
- $\overrightarrow{r}, \Xi.\mathsf{pos}(\overrightarrow{r}) \simeq_\sigma \hat{\overrightarrow{r}}, \hat{\Xi}.\mathsf{pos}(\hat{\overrightarrow{r}})$ **(4)** - (2) + (3)

In the rest of the proof we assume that $\Xi(r).\mathsf{pos} \sqcup \hat{\Xi}(r).\mathsf{pos} \sqsubseteq \sigma$ (hyp.7) and we proceed by induction on the structure of the derivation of $f \vdash r \rightsquigarrow_m \overrightarrow{r}$. There are two base cases to consider [Node Not Found - Leaf Node] and [Node Found - Leaf Node]. The inductive cases are [Node Not Found - Non-Leaf Node] and [Node Found - Non-Leaf Node]. Since both the base cases and the inductive cases are analogous, we only consider the cases [Node Not Found - Leaf Node] and [Node Found - Non-Leaf Node].

[Node Not Found - Leaf Node] In this case: $|f(r).\mathsf{children}| = 0$ and $f(r).\mathsf{tag} \neq m$ (hyp.8). Letting $\hat{\overrightarrow{r}}_i$ be: $\hat{f} \vdash \hat{f}(r).\mathsf{children}(i) \rightsquigarrow_m \hat{\overrightarrow{r}}_i$, for $0 \leq i < |\hat{f}(r).\mathsf{children}|$, we conclude that:

- $\overrightarrow{r} = \varepsilon$ (1) - (hyp.3) + (hyp.8)
- $\hat{f}(r).\mathsf{tag} \neq m$ (2) - (hyp.5) + (hyp.7) + (hyp.8)
- $\forall_{0 \leq i < |\hat{f}(r).\mathsf{children}|}\ \hat{\Xi}(\hat{f}(r).\mathsf{children}(i)).\mathsf{pos} \not\sqsubseteq \sigma$ (3) - (hyp.5) + (hyp.7)
- For all $0 \leq i < |\hat{f}(r).\mathsf{children}|$ and for all $0 \leq j < |\overrightarrow{\hat{r}}_i|$: $\hat{\Xi}(\overrightarrow{\hat{r}}_i(j)) \not\sqsubseteq \sigma$ (4) - (3) + *Monotonocity of Search Predicate*
- $\overrightarrow{r}, \Xi.\mathsf{pos}(\overrightarrow{r}) \simeq_\sigma \overrightarrow{\hat{r}}, \hat{\Xi}.\mathsf{pos}(\overrightarrow{\hat{r}})$ **(5)** - (1) + (2) + (4)

[Node Found - Non-Leaf Node] In this case: $|f(r).\mathsf{children}| = n > 0$ and $f(r).\mathsf{tag} = m$ (hyp.8). Without loss of generality, let us assume that $|\hat{f}(r).\mathsf{children}| = \hat{n} > 0$ (hyp.9). We conclude that:

- $\overrightarrow{r} = r :: \overrightarrow{r}_0 :: \cdots :: \overrightarrow{r}_{n-1}$, where $f \vdash f(r).\mathsf{children}(i) \rightsquigarrow_m \overrightarrow{r}_i$ for $0 \leq i < n$ (1) - (hyp.3) + (hyp.8)
- $\overrightarrow{\hat{r}} = r :: \overrightarrow{\hat{r}}_0 :: \cdots :: \overrightarrow{\hat{r}}_{\hat{n}-1}$, where $\hat{f} \vdash \hat{f}(r).\mathsf{children}(i) \rightsquigarrow_m \overrightarrow{\hat{r}}_i$ for $0 \leq i < \hat{n}$ (2) - (hyp.4) + (hyp.9)
- $\forall_{0 \leq i < n}\ \mathcal{S}ec_{f,\Xi} \vdash^{f(r).\mathsf{children}(i)} \phi^i_t, \varphi^i_t \rightsquigarrow \phi^{i+1}_t, \varphi^{i+1}_t$ and $\phi'_t = \phi^n_t$ (3) - (hyp.1) + (hyp.8)
- $\forall_{0 \leq i < \hat{n}}\ \mathcal{S}ec_{\hat{f},\hat{\Xi}} \vdash^{\hat{f}(r).\mathsf{children}(i)} \hat{\phi}^i_t, \hat{\varphi}^i_t \rightsquigarrow \hat{\phi}^{i+1}_t, \hat{\varphi}^{i+1}_t$ and $\hat{\phi}'_t = \hat{\phi}^{\hat{n}}_t$ (4) - (hyp.2) + (hyp.9)

Let i be the largest integer such that $\phi^{i-1}_t(m) \sqsubseteq \sigma$ and $\phi^i_t(m) \not\sqsubseteq \sigma$ and let j be the largest integer such that $\hat{\phi}^{j-1}_t(m) \sqsubseteq \sigma$ and $\hat{\phi}^j_t(m) \not\sqsubseteq \sigma$. We have to prove that:

1. Prove that i and j coincide.
2. Prove that for every integer $0 \leq l < i = j$, it holds that:
$$r :: \overrightarrow{r}_0 :: \cdots :: \overrightarrow{r}_l, \Xi(r).\mathsf{pos} :: \Xi.\mathsf{pos}(\overrightarrow{r}_0) :: \cdots :: \Xi.\mathsf{pos}(\overrightarrow{r}_l) \approx_\sigma r :: \overrightarrow{\hat{r}}_0 :: \cdots :: \overrightarrow{\hat{r}}_l, \hat{\Xi}(r).\mathsf{pos} :: \hat{\Xi}.\mathsf{pos}(\overrightarrow{\hat{r}}_0) :: \cdots :: \hat{\Xi}.\mathsf{pos}(\overrightarrow{\hat{r}}_l)$$
3. Prove that:
$$r :: \overrightarrow{r}_0 :: \cdots :: \overrightarrow{r}_i, \Xi(r).\mathsf{pos} :: \Xi.\mathsf{pos}(\overrightarrow{r}_0) :: \cdots :: \Xi.\mathsf{pos}(\overrightarrow{r}_i) \simeq_\sigma r :: \overrightarrow{\hat{r}}_0 :: \cdots :: \overrightarrow{\hat{r}}_i, \hat{\Xi}(r).\mathsf{pos} :: \hat{\Xi}.\mathsf{pos}(\overrightarrow{\hat{r}}_0) :: \cdots :: \hat{\Xi}.\mathsf{pos}(\overrightarrow{\hat{r}}_i)$$
4. Prove that: $\sqcap\{\sqcap\Xi.\mathsf{pos}(\overrightarrow{r}_l) \mid i < l < |f(r).\mathsf{children}|\} \not\sqsubseteq \sigma$
5. Prove that: $\sqcap\{\sqcap\hat{\Xi}.\mathsf{pos}(\overrightarrow{\hat{r}}_l) \mid i < l < |\hat{f}(r).\mathsf{children}|\} \not\sqsubseteq \sigma$

Proof of 1. Suppose that $\phi^{i-1}_t(m) \sqsubseteq \sigma$ and $\phi^i_t(m) \not\sqsubseteq \sigma$ and let j be the largest integer such that $\hat{\phi}^{j-1}_t(m) \sqsubseteq \sigma$ and $\hat{\phi}^j_t(m) \not\sqsubseteq \sigma$ (hyp.10). We conclude that:

- $\varphi^{i-1}_t \restriction^\sigma = \hat{\varphi}^{i-1}_t \restriction^\sigma$ and $\varphi^i_t \restriction^\sigma = \hat{\varphi}^i_t \restriction^\sigma$ (5) - (hyp.1) + (hyp.2) + (hyp.5) + (hyp.6) + Lemma D.3
- $\hat{\phi}^{i-1}_t(m) \sqsubseteq \sigma$ (6) - (hyp.10) + (5)
- $\hat{\phi}^i_t(m) \not\sqsubseteq \sigma$ (7) - (hyp.10) + (5)
- $j = i$ (8) - (6) + (7)

Proof of 2. We proceed by induction on l.

Base case: $l = 0$.

- $\overrightarrow{r}_0, \Xi.\mathsf{pos}(\overrightarrow{r}_0) \simeq_\sigma \overrightarrow{\hat{r}}_0, \Xi.\mathsf{pos}(\overrightarrow{r}_0)$ (9) - (hyp.5) + (hyp.6) + (1)-(4) + **outer ih**

- $\overrightarrow{r}_0, \Xi.\mathsf{pos}(\overrightarrow{r}_0) \approx_\sigma \overrightarrow{\hat{r}}_0, \Xi.\mathsf{pos}(\overrightarrow{r}_0)$ (10) - (hyp.10) + (9)

Inductive case: $l = l' + 1$.

- $r :: \overrightarrow{r}_0 :: \cdots :: \overrightarrow{r}'_l, \Xi(r).\mathsf{pos} :: \Xi.\mathsf{pos}(\overrightarrow{r}_0) :: \cdots :: \Xi.\mathsf{pos}(\overrightarrow{r}'_l) \approx_\sigma$ $r :: \overrightarrow{\hat{r}}_0 :: \cdots :: \overrightarrow{\hat{r}}'_l, \hat{\Xi}(r).\mathsf{pos} :: \hat{\Xi}.\mathsf{pos}(\overrightarrow{\hat{r}}_0) :: \cdots :: \hat{\Xi}.\mathsf{pos}(\overrightarrow{\hat{r}}'_l)$ (11) - **inner ih**
- $\overrightarrow{r}_l, \Xi.\mathsf{pos}(\overrightarrow{r}_l) \simeq_\sigma \overrightarrow{\hat{r}}_l, \Xi.\mathsf{pos}(\overrightarrow{r}_l)$ (12) - (hyp.5) + (hyp.6) + (1)-(4) + **outer ih**
- $\overrightarrow{r}_l, \Xi.\mathsf{pos}(\overrightarrow{r}_l) \approx_\sigma \overrightarrow{\hat{r}}_l, \Xi.\mathsf{pos}(\overrightarrow{r}_l)$ (13) - (hyp.10) + (12)
- $r :: \overrightarrow{r}_0 :: \cdots :: \overrightarrow{r}_l, \Xi(r).\mathsf{pos} :: \Xi.\mathsf{pos}(\overrightarrow{r}_0) :: \cdots :: \Xi.\mathsf{pos}(\overrightarrow{r}_l) \approx_\sigma$ $r :: \overrightarrow{\hat{r}}_0 :: \cdots :: \overrightarrow{\hat{r}}_l, \hat{\Xi}(r).\mathsf{pos} :: \hat{\Xi}.\mathsf{pos}(\overrightarrow{\hat{r}}_0) :: \cdots :: \hat{\Xi}.\mathsf{pos}(\overrightarrow{\hat{r}}_l)$ (14) - (11) + (13)

Proof of 3.

- $\overrightarrow{r}_i, \Xi.\mathsf{pos}(\overrightarrow{r}_i) \simeq_\sigma \overrightarrow{\hat{r}}_i, \Xi.\mathsf{pos}(\overrightarrow{r}_i)$ (15) - (hyp.5) + (hyp.6) + (1)-(4) + **ih**
- $r :: \overrightarrow{r}_0 :: \cdots :: \overrightarrow{r}_i, \Xi(r).\mathsf{pos} :: \Xi.\mathsf{pos}(\overrightarrow{r}_0) :: \cdots :: \Xi.\mathsf{pos}(\overrightarrow{r}_i) \simeq_\sigma$ $r :: \overrightarrow{\hat{r}}_0 :: \cdots :: \overrightarrow{\hat{r}}_i, \hat{\Xi}(r).\mathsf{pos} :: \hat{\Xi}.\mathsf{pos}(\overrightarrow{\hat{r}}_0) :: \cdots :: \hat{\Xi}.\mathsf{pos}(\overrightarrow{\hat{r}}_i)$ (16) - (14) + (15)

Proof of 4.

- $\forall_{i<l<|f(r).\mathsf{children}|}\ \varphi^l_\natural(m,\sigma) = \varphi^{l+1}_\natural(m,\sigma)$ (17) - (hyp.1) + (hyp.10)
- $\forall_{i<l<|f(r).\mathsf{children}|}\ \forall_{0\leq k<|\overrightarrow{r}_l|}\ \Xi(\overrightarrow{r}_l(k)).\mathsf{pos} \not\sqsubseteq \sigma$ (18) - (hyp.1) + (17) + Lemma D.5

Proof of 5.

- $\forall_{i<l<|\hat{f}(r).\mathsf{children}|}\ \hat{\varphi}^l_\natural(m,\sigma) = \hat{\varphi}^{l+1}_\natural(m,\sigma)$ (19) - (hyp.2) + (hyp.10) + (8)
- $\forall_{i<l<|\hat{f}(r).\mathsf{children}|}\ \forall_{0\leq k<|\overrightarrow{\hat{r}}_l|}\ \hat{\Xi}(\overrightarrow{\hat{r}}_l(k)).\mathsf{pos} \not\sqsubseteq \sigma$ (20) - (hyp.2) + (19) + Lemma D.5

□

Theorem 7.2- Low-Equality Strengthening

Proof: Given that:

- $\mathcal{S}ec(f_0, \Xi_0)$ (hyp.1)
- $\mathcal{S}ec(f_1, \Xi_1)$ (hyp.1)
- $f_0, \Xi_0 \sim^\sigma_{DOM} f_1, \Xi_1$ (hyp.3),

We have to prove that: $f_0, \Xi_0 \sim^\sigma_\natural f_1, \Xi_1$. In order to prove this, we have to prove that if $(r, m, i, r') \in f_0 \restriction^{\Xi_0,\sigma}_\natural$, then $(r, m, i, r') \in f_1 \restriction^{\Xi_1,\sigma}_\natural$ and that if $(r, m, n) \in f_0 \restriction^{\Xi_0,\sigma}_\natural$, then $(r, m, n) \in f_1 \restriction^{\Xi_1,\sigma}_\natural$.

Suppose that: $(r, m, i, r') \in f_0 \restriction^{\Xi_0,\sigma}_\natural$ (hyp.4). We conclude that:

- $f_0 \vdash r \rightsquigarrow_m \overrightarrow{r}_0$, $\overrightarrow{r}_0(i) = r'$, and $\Xi_0(r').\mathsf{pos} \sqsubseteq \sigma$ (1) - (hyp.4)
- $\Xi_0(r).\mathsf{pos} \sqsubseteq \sigma$ (2) - (hyp.1) + (1)
- $\Xi_0(r).\mathsf{node} \sqsubseteq \sigma$ (3) - (2)
- $r \in dom(f_1)$, $\Xi_1(r).\mathsf{node} \sqsubseteq \sigma$, and $\Xi_1(r).\mathsf{pos} \sqsubseteq \sigma$ (4) - (hyp.3) + (2)

If we let $\overrightarrow{r}_1$ be the list of nodes verifying $f_1 \vdash r \rightsquigarrow_m \overrightarrow{r}_1$ (hyp.5), we conclude that:

- $\overrightarrow{r}_0, \Xi_0.\mathsf{pos}(\overrightarrow{r}_0) \simeq_\sigma \overrightarrow{r}_1, \Xi_1.\mathsf{pos}(\overrightarrow{r}_1)$ (5) - (hyp.1)-(hyp.5) + Lemma D.6
- $\overrightarrow{r}_1(i) = r'$ and $\Xi_1(r').\mathsf{pos} \sqsubseteq \sigma$ (6) - (1) + (5)
- $(r, m, i, r') \in f_1 \restriction^{\Xi_1,\sigma}_\natural$ **(7)** - (hyp.5) + (6)

Suppose that: $(r, m, n) \in f_0 \upharpoonright_{\lightning}^{\Xi_0,\sigma}$ (hyp.4). We conclude that:

- $f_0 \vdash r \rightsquigarrow_m \overrightarrow{r}_0$, $|\overrightarrow{r}_0| = n$, and $\sigma_m \sqcup \Xi(r).\mathsf{node} \sqsubseteq \sigma$ (1) - (hyp.4)
- $r \in dom(f_1)$ and $\Xi_1(r).\mathsf{node} \sqsubseteq \sigma$ (2) - (hyp.3) + (1)

If we let $\overrightarrow{r}_1$ be the list of nodes verifying $f_1 \vdash r \rightsquigarrow_m \overrightarrow{r}_1$ (hyp.5), we conclude that:

- $\overrightarrow{r}_0, \Xi_0.\mathsf{pos}(\overrightarrow{r}_0) \simeq_\sigma \overrightarrow{r}_1, \Xi_1.\mathsf{pos}(\overrightarrow{r}_1)$ (3) - (hyp.1)-(hyp.5) + Lemma D.6
- $\sqcup(\Xi_0.\mathsf{pos}(\overrightarrow{r}_0)) \sqsubseteq \sigma_m$ (4) - (hyp.1) + (hyp.4)
- $\sqcup(\Xi_1.\mathsf{pos}(\overrightarrow{r}_1)) \sqsubseteq \sigma_m$ (5) - (hyp.2) + (hyp.5)
- $|\overrightarrow{r}_0| = |\overrightarrow{r}_1|$ (6) - (3)-(5)
- $(r, m, n) \in f_1 \upharpoonright_{\lightning}^{\Xi_1,\sigma}$ **(7)** - (hyp.5) + (6)

□

D.3 Noninterference - Live Collections Monitor

Lemma 7.3 - Confinement - Monitored Core DOM + Live Collections

Proof: We proceed by case analysis. We only consider the monitored plugins that can change the memory: $(\mathsf{new}_{\lightning}, \mathsf{new}_{lab}^{\lightning})$ and $(\mathsf{redirect}_{\lightning}, \mathsf{redirect}_{lab}^{\lightning})$.

[Core DOM Redirection] Given that:

- $\langle \nu, r_0 :: v_1 :: \overrightarrow{v}' \rangle\ \mathsf{redirect}_{\lightning}\ \langle \langle f', \nu.lives \rangle, v \rangle^{\beta}$ (hyp.1)
- $\langle \Xi, \sigma_0 :: \sigma_1 :: \overrightarrow{\sigma}' \rangle^{(r_0, v_1, \beta)}\ \mathsf{redirect}_{lab}^{\lightning}\ \langle \langle \Xi', \Xi.lives \rangle, \sigma' \rangle$ (hyp.2)
- $\overrightarrow{\sigma}(0) \sqcup \overrightarrow{\sigma}(1) \not\sqsubseteq \sigma$ (hyp.3)

We conclude that:

- $(\mathsf{dplug}, \mathsf{dplug}_{lab}) = \mathcal{R}_{IF}^{DOM}(r_0, v_1)$ (1) - (hyp.1) + (hyp.2)
- $\langle \nu.f, r_0 :: v_1 :: \overrightarrow{v}' \rangle\ \mathsf{dplug}\ \langle f', v \rangle^{\beta}$ (2) - (hyp.1) + (hyp.2)
- $\langle \Xi.f, \sigma_0 :: \sigma_1 :: \overrightarrow{\sigma}' \rangle^{\beta}\ \mathsf{dplug}_{lab}\ \langle \Xi', \sigma' \rangle$ (3) - (hyp.1) + (hyp.2)
- $f, \Xi \sim_{DOM}^{\sigma} f', \Xi'$ (4) - (hyp.3) + (2) + (3) + Confinement (Lemma 7.2)
- $\nu, \Xi\ \sim_{DOM}^{\sigma}\ \langle f', \nu.lives \rangle, \langle \Xi', \Xi.lives \rangle$ **(5)** - (2) - (4)

[Live New] Given that:

- $\langle \nu, r :: \texttt{"getElementByTagName"} :: m \rangle^{\sigma_l}\ \mathsf{new}_{\lightning}\ \langle \nu', r' \rangle^{(r', \sigma_l)}$ (hyp.1)
- $\langle \Xi, \sigma_0 :: \sigma_1 :: \sigma_2 \rangle^{r}\ \mathsf{new}_{lab}^{\lightning}\ \langle \Xi', \sigma_l \rangle$ (hyp.2)
- $\sigma_0 \sqcup \sigma_1 \not\sqsubseteq \sigma$ (hyp.3)

We conclude that:

- $r' = \mathsf{fresh}_{live}(\sigma_l)$, $lives' = \nu.lives\,[r' \mapsto \langle r, m \rangle]$, and $\nu' = \langle \nu.f, lives' \rangle$ (1) - (hyp.1)
- $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \sigma_l$ and $\Xi' = \langle \Xi.f, \Xi.lives\,[r \mapsto \sigma_l] \rangle$ (2) - (hyp.2)
- $\sigma_l \not\sqsubseteq \sigma$ (3) - (hyp.3) + (2)
- $\nu, \Xi\ \sim_{DOM}^{\sigma}\ \nu', \Xi'$ **(4)** - (1) - (3)

Theorem 7.3 - Noninterference - Monitored Core DOM + Live Collections

Proof: For every $(\mathsf{pg}, \mathsf{pg}_{lab})$ in the range of $\mathcal{R}^{\natural}_{IF}$, we have to prove that given two DOM states ν and ν' labelled by Ξ and Ξ' and two sequences of values $\overrightarrow{v}$ and $\overrightarrow{v}'$ respectively labelled by two sequences of levels $\overrightarrow{\sigma}$ and $\overrightarrow{\sigma}'$ and such that:

- $\overrightarrow{v}, \overrightarrow{\sigma} \sim_{\sigma} \overrightarrow{v}, \overrightarrow{\sigma}'$ (hyp.1)
- $\nu, \Xi \sim^{\sigma}_{DOM} \nu', \Xi'$ (hyp.2)
- $\langle \nu, \overrightarrow{v} \rangle^{\alpha}\ \mathsf{pg}\ \langle \nu_f, v_f \rangle^{\beta}$ (hyp.3) and $\langle \nu', \overrightarrow{v}' \rangle^{\alpha}\ \mathsf{pg}\ \langle \nu'_f, v'_f \rangle^{\beta'}$ (hyp.4)
- $\langle \Xi, \overrightarrow{\sigma} \rangle^{\beta}\ \mathsf{pg}_{lab}\ \langle \Xi_f, \sigma_f \rangle$ (hyp.5) and $\langle \Xi', \overrightarrow{\sigma}' \rangle^{\beta'}\ \mathsf{pg}_{lab}\ \langle \Xi'_f, \sigma'_f \rangle$ (hyp.6)

Then, it holds that: $\nu_f, \Xi_f \sim^{\sigma}_{DOM} \nu'_f, \Xi'_f$ and $v_f, \sigma_f \sim_{\sigma} v'_f, \sigma'_f$. In order to prove that $v_f, \sigma_f \sim_{\sigma} v'_f, \sigma'_f$, we have to prove the following two implications:

- $\sigma_f \sqsubseteq \sigma \Rightarrow v_f = v'_f \wedge \sigma_f = \sigma'_f \sqsubseteq \sigma$
- $\sigma'_f \sqsubseteq \sigma \Rightarrow v_f = v'_f \wedge \sigma_f = \sigma'_f \sqsubseteq \sigma$.

Since the proofs are identical, we only prove first one. However, we cannot introduce at this level the hypothesis $\sigma_f \sqsubseteq \sigma$ because it cannot be used in the proof of $\nu_f, \Xi_f \sim_{\sigma} \nu'_f, \Xi'_f$. Therefore, we are obliged to introduce this hypothesis in every case. We now proceed by case analysis on the monitored plugins in the range of $\mathcal{R}^{\natural}_{IF}$.

[Live New] Suppose $(\mathsf{pg}, \mathsf{pg}_{lab}) = (\mathsf{new}_{\natural}, \mathsf{new}^{\natural}_{lab})$ (hyp.7). We conclude that:

- $\overrightarrow{v} = r :: \_ :: m$, $\overrightarrow{v}' = r' :: \_ :: m'$, $\overrightarrow{\sigma} = \sigma_0 :: \sigma_1 :: \sigma_2$, $\overrightarrow{\sigma}' = \sigma'_0 :: \sigma'_1 :: \sigma'_2$, and $\alpha = \sigma_l$ (1) - (hyp.3) - (hyp.7)
- $v_f = r_f = \mathsf{fresh}_{live}(\sigma_l)$ and $v'_f = r'_f = \mathsf{fresh}_{live}(\sigma_l)$ (2) - (hyp.3) + (hyp.4) + (hyp.7)
- $\nu_f = \langle \nu.f, lives_f \rangle$ where $lives_f = \nu.lives\,[r_f \mapsto \langle r, m \rangle]$ (3) - (hyp.3) + (hyp.7)
- $\nu'_f = \langle \nu'.f, lives'_f \rangle$ where $lives'_f = \nu'.lives\left[r'_f \mapsto \langle r', m' \rangle\right]$ (4) - (hyp.4) + (hyp.7)
- $\Xi_f = \langle \Xi.f, \Xi.lives\,[r_f \mapsto \sigma_l] \rangle$ and $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \sigma_l$ (5) - (hyp.5) + (hyp.7)
- $\Xi'_f = \langle \Xi'.f, \Xi'.lives\left[r'_f \mapsto \sigma_l\right] \rangle$ and $\sigma'_0 \sqcup \sigma'_1 \sqcup \sigma'_2 \sqsubseteq \sigma_l$ (6) - (hyp.6) + (hyp.7)
- $\nu.f\ \restriction^{\Xi.f,\sigma}_{\natural} = \nu'.f\ \restriction^{\Xi'.f,\sigma}_{\natural}$ and $\nu.lives\ \restriction^{\Xi.lives,\sigma}_{\natural} = \nu'.lives\ \restriction^{\Xi'.lives,\sigma}_{\natural}$ (7) - (hyp.2)
- $\nu.f = \nu_f.f$, $\Xi.f = \Xi_f.f$, $\nu'.f = \nu'_f.f$, and $\Xi'.f = \Xi'_f.f$ (8) - (3) - (6)
- $\nu_f.f\ \restriction^{\Xi_f.f,\sigma}_{\natural} = \nu'_f.f\ \restriction^{\Xi'_f.f,\sigma}_{\natural}$ (9) - (7) + (8)

There are two cases to consider: either $\sigma_l \sqsubseteq \sigma$ or $\sigma_l \not\sqsubseteq \sigma$. Suppose $\sigma_l \sqsubseteq \sigma$ (hyp.8):

- $r_f = r'_f$ (10) - (hyp.2) + (hyp.8) + (2) + Low-equal Allocation
- $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \sqsubseteq \sigma$ (11) - (hyp.8) + (5)
- $r = r'$, $m = m'$, $\sigma'_0 = \sigma_0 \sqsubseteq \sigma$, $\sigma_1 = \sigma'_1 \sqsubseteq \sigma$, and $\sigma_2 = \sigma'_2 \sqsubseteq \sigma$ (12) - (hyp.1) + (10)
- $lives_f\ \restriction^{\Xi_f.lives,\sigma}_{\natural} = lives\ \restriction^{\Xi.lives,\sigma}_{\natural} \cup \{(r_f, r, m, \sigma_l)\}$ (13) - (3) + (5) + (11)
- $lives'_f\ \restriction^{\Xi'_f.lives,\sigma}_{\natural} = lives'\ \restriction^{\Xi'.lives,\sigma}_{\natural} \cup \{(r'_f, r', m', \sigma_l)\}$ (14) - (4) + (6) + (12)
- $\{(r_f, r, m, \sigma_l)\} = \{(r'_f, r', m', \sigma_l)\}$ (15) - (10) + (12)

- $lives_f \restriction_{\lightning}^{\Xi_f.lives,\sigma} = lives'_f \restriction_{\lightning}^{\Xi'_f.lives,\sigma}$ (16) - (8) + (13) - (15)
- $\nu_f \restriction_{\lightning}^{\Xi_f,\sigma} = \nu'_f \restriction_{\lightning}^{\Xi'_f,\sigma}$ **(17)** - (10) + (16)
- $v_f = v'_f$ and $\sigma_f = \sigma'_f$ **(18)** - (7)

Suppose $\sigma_l \not\sqsubseteq \sigma$ (hyp.8):

- $lives_f \restriction_{\lightning}^{\Xi_f.lives,\sigma} = lives \restriction_{\lightning}^{\Xi.lives,\sigma}$ (19) - (hyp.8) + (3) + (5)
- $lives'_f \restriction_{\lightning}^{\Xi'_f.lives,\sigma} = lives' \restriction_{\lightning}^{\Xi'.lives,\sigma}$ (20) - (hyp.8) + (4) + (6)
- $\nu_f \restriction_{\lightning}^{\Xi_f,\sigma} = \nu'_f \restriction_{\lightning}^{\Xi'_f,\sigma}$ **(21)** - (hyp.2) + (9) + (19) + (20)

[Live Length] Suppose $(\mathsf{pg}, \mathsf{pg}_{lab}) = (\mathsf{length}_{\lightning}, \mathsf{length}_{lab}^{\lightning})$ (hyp.7). We conclude that:

- $\overrightarrow{v} = r :: \_$, $\overrightarrow{v}' = r' :: \_$, $\overrightarrow{\sigma} = \sigma_0 :: \sigma_1$, and $\overrightarrow{\sigma}' = \sigma'_0 :: \sigma'_1$ (1) - (hyp.3) - (hyp.7)
- $\nu_f = \nu$, $\nu'_f = \nu'$, $\Xi_f = \Xi$, and $\Xi'_f = \Xi'$ (2) - (hyp.3) + (hyp.7)
- $\nu_f, \Xi_f \sim_{DOM}^{\sigma} \nu'_f, \Xi'_f$ **(3)** - (hyp.2) + (2)
- $v_f = |\overrightarrow{r}|$ where: $\nu.lives(r) = \langle \hat{r}, m \rangle$ and $\nu.f \vdash \hat{r} \rightsquigarrow_m \overrightarrow{r}$ (4) - (hyp.3) + (hyp.7)
- $v'_f = |\overrightarrow{r}'|$ where: $\nu'.lives(r') = \langle \hat{r}', m' \rangle$ and $\nu'.f \vdash \hat{r}' \rightsquigarrow_{m'} \overrightarrow{r}'$ (5) - (hyp.4) + (hyp.7)
- $\sigma_f = \sigma_0 \sqcup \sigma_1 \sqcup \Xi.lives(r) \sqcup \sigma_m$ (6) - (hyp.5) + (hyp.7)
- $\sigma'_f = \sigma'_0 \sqcup \sigma'_1 \sqcup \Xi'.lives(r') \sqcup \sigma_{m'}$ (7) - (hyp.6) + (hyp.7)
- $\mathcal{S}ec(\nu.f, \Xi.f, \hat{r})$ and $\mathcal{S}ec(\nu'.f, \Xi'.f, \hat{r}')$ (8) - (hyp.5) + (hyp.6) + (hyp.7)
- $\nu.f, \Xi.f \sim_{\lightning}^{\sigma} \nu'.f, \Xi'.f$ (9) - (hyp.2) + (8) + *Low-Equality Strengthening* (Theorem 7.2)

Suppose that $\sigma_f \sqsubseteq \sigma$ (hyp.8):

- $\sigma_0 \sqcup \sigma_1 \sqcup \Xi.lives(r) \sqcup \sigma_m \sqsubseteq \sigma$ (10) - (hyp.8) + (6)
- $r = r'$, $\sigma'_0 = \sigma_0 \sqsubseteq \sigma$, and $\sigma_1 = \sigma'_1 \sqsubseteq \sigma$ (11) - (hyp.1) + (1) + (10)
- $\Xi.lives(r) = \Xi'.lives(r') \sqsubseteq \sigma$ and $\langle \hat{r}, m \rangle = \langle \hat{r}', m' \rangle$ (12) - (hyp.2) + (10) + (11)
- $v_f = |\overrightarrow{r}| = |\overrightarrow{r}'| = v'_f$ **(13)** - (hyp.2) + (4) + (5) + (9) + (11) + (12)

[Live Item] Suppose $(\mathsf{pg}, \mathsf{pg}_{lab}) = (\mathsf{item}_{\lightning}, \mathsf{item}_{lab}^{\lightning})$ (hyp.7). We conclude that:

- $\overrightarrow{v} = r :: i$, $\overrightarrow{v}' = r' :: i''$, $\overrightarrow{\sigma} = \sigma_0 :: \sigma_1$, and $\overrightarrow{\sigma}' = \sigma'_0 :: \sigma'_1$ (1) - (hyp.3) - (hyp.7)
- $\nu_f = \nu$ and $\nu'_f = \nu'$ (2) - (hyp.3) + (hyp.7)
- $\nu, \Xi \sim_{DOM}^{\sigma} \nu', \Xi'$ **(3)** - (hyp.2) + (3)
- $v_f = r_f = \overrightarrow{r}(i)$ where: $\nu.lives(r) = \langle \hat{r}, m \rangle$ and $\nu.f \vdash \hat{r} \rightsquigarrow_m \overrightarrow{r}$ (4) - (hyp.3) + (hyp.7)
- $v'_f = r'_f = \overrightarrow{r}'(i')$ where: $\nu'.lives(r') = \langle \hat{r}', m' \rangle$ and $\nu'.f \vdash \hat{r}' \rightsquigarrow_{m'} \overrightarrow{r}'$ (5) - (hyp.4) + (hyp.7)
- $\sigma_f = \sigma_0 \sqcup \sigma_1 \sqcup \Xi.lives(r) \sqcup \Xi.f(r_f).\mathsf{pos}$ (6) - (hyp.5) + (hyp.7)
- $\sigma'_f = \sigma'_0 \sqcup \sigma'_1 \sqcup \Xi'.lives(r') \sqcup \Xi'.f(r'_f).\mathsf{pos}$ (7) - (hyp.6) + (hyp.7)
- $\mathcal{S}ec(\nu.f, \Xi.f, \hat{r})$ and $\mathcal{S}ec(\nu'.f, \Xi'.f, \hat{r}')$ (8) - (hyp.5) + (hyp.6) + (hyp.7)
- $\nu.f, \Xi.f \sim_{\lightning}^{\sigma} \nu'.f, \Xi'.f$ (9) - (hyp.2) + (8) + *Low-Equality Strengthening* (Theorem 7.2)

Suppose that $\sigma_f \sqsubseteq \sigma$ (hyp.8):

- $\sigma_0 \sqcup \sigma_1 \sqcup \Xi.lives(r) \sqcup \Xi.f(r_f).\mathsf{pos} \sqsubseteq \sigma$ (10) - (hyp.8) + (6)
- $r = r'$, $i = i'$, $\sigma'_0 = \sigma_0 \sqsubseteq \sigma$, and $\sigma_1 = \sigma'_1 \sqsubseteq \sigma$ (11) - (hyp.1) + (1) + (10)

- $\Xi.lives(r) = \Xi'.lives(r') \sqsubseteq \sigma$ and $\langle \hat{r}, m\rangle = \langle \hat{r}', m'\rangle$ (12) - (hyp.2) + (9) + (10)
- $v_f = \overrightarrow{r}(i) = \overrightarrow{r}'(i') = v'_f$ **(13)** - (hyp.2) + (5) + (6) + (9) - (12)

[Core DOM Redirection] Suppose $(\mathsf{pg}, \mathsf{pg}_{lab}) = (\mathsf{redirect}_{\lightning}, \mathsf{redirect}^{\lightning}_{lab})$ (hyp.7). We conclude that:

- $(\mathsf{dplug}, \mathsf{dplug}_{lab}) = \mathcal{R}^{DOM}_{IF}(\overrightarrow{v}(0), \overrightarrow{v}(1))$ (1) - (hyp.3) + (hyp.7)
- $(\mathsf{dplug}', \mathsf{dplug}'_{lab}) = \mathcal{R}^{DOM}_{IF}(\overrightarrow{v}'(0), \overrightarrow{v}'(1))$ (2) - (hyp.4) + (hyp.7)
- $\langle \nu.f, \overrightarrow{v}\rangle$ dplug $\langle f_f, v_f\rangle^\beta$ and $\langle \Xi.f, \overrightarrow{\sigma}\rangle^\beta$ dplug_{lab} $\langle \Xi_f, \sigma_f\rangle$ (3) - (hyp.3) + (hyp.5) + (hyp.7)
- $\langle \nu'.f, \overrightarrow{v}'\rangle$ dplug' $\langle f'_f, v'_f\rangle^\beta$ and $\langle \Xi'.f, \overrightarrow{\sigma}'\rangle^\beta$ dplug_{lab} $\langle \Xi'_f, \sigma'_f\rangle$ (4) - (hyp.4) + (hyp.6) + (hyp.7)

We consider two distinct cases $\overrightarrow{v}(0) \sqcup \overrightarrow{v}(1) \sqsubseteq \sigma$ and $\overrightarrow{v}(0) \sqcup \overrightarrow{v}(1) \not\sqsubseteq \sigma$. Suppose that $\overrightarrow{v}(0) \sqcup \overrightarrow{v}(1) \sqsubseteq \sigma$ (hyp.8). We conclude that:

- $\overrightarrow{v}'(0) = \overrightarrow{v}(0)$ and $\overrightarrow{v}'(1) = \overrightarrow{v}(1)$ (5) - (hyp.1) + (hyp.8)
- $(\mathsf{dplug}, \mathsf{dplug}_{lab}) = (\mathsf{dplug}', \mathsf{dplug}'_{lab})$ (6) - (1) + (2) + (5)
- $f_f, \Xi_f \sim^\sigma_{DOM} f'_f, \Xi'_f$ and $v_f, \sigma_f \sim_\sigma v'_f, \sigma'_f$ (7) - (hyp.1) + (hyp.2) + (3) + (4) + *Noninterferent Core DOM API*
- $\langle f_f, \nu.lives\rangle, \langle \Xi_f, \Xi.lives\rangle \sim^\sigma_{DOM} \langle f'_f, \nu'.lives\rangle, \langle \Xi'_f, \Xi'.lives\rangle$ **(8)** - (hyp.3)-(hyp.6) + (3) + (4) + (7)

Suppose that $\overrightarrow{v}(0) \sqcup \overrightarrow{v}(1) \not\sqsubseteq \sigma$ (hyp.8). We conclude that:

- $\overrightarrow{v}'(0) \sqcup \overrightarrow{v}'(1) \not\sqsubseteq \sigma$ (9) - (hyp.1) + (hyp.8)
- $\nu, \Xi \sim^\sigma_{DOM} \nu_f, \Xi_f$ (10) - (hyp.3) + (hyp.5) + (hyp.8)
- $\nu', \Xi' \sim^\sigma_{DOM} \nu'_f, \Xi'_f$ (11) - (hyp.3) + (hyp.5) + (9)
- $\nu_f, \Xi_f \sim^\sigma_{DOM} \nu'_f, \Xi'_f$ **(12)** - (hyp.2)-(hyp.6) + (10) + (11)

□