Tipos para seguridad basada en flujo de información y computación auditada

Bavera, Francisco Pedro

2012

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Tipos para seguridad basada en flujo de información y computación auditada

Tesis presentada para optar al título de Doctor de la Universidad de Buenos Aires en el área Ciencias de la Computación

Francisco Pedro Bavera

Director: Eduardo Bonelli

Lugar de Trabajo: Departamento de Computación, Facultad de Ciencias Exactas, Físico-Químicas y Naturales, Universidad Nacional de Río Cuarto

Buenos Aires - 2012

Tipos para Seguridad basada en Flujo de Información y Computación Auditada

Resumen

Garantizar la confidencialidad de la información restringiendo su acceso sólo a usuarios autorizados es uno de los pilares de la seguridad informática actual. Sin embargo, una vez otorgado dicho acceso no se conserva control alguno sobre cómo se utiliza. El Análisis del Flujo de Información (IFA) estudia cómo se transmite la información por diferentes mecanismos. Para ello se vale de políticas de flujo de información previamente establecidas. IFA vela por el cumplimiento de las mismas, a través de técnicas de análisis estático de programas o a través de monitores de ejecución. De esta manera se logra un control sobre el uso de los datos, que aplica más allá del momento en que se otorga el acceso a los mismos. Una alternativa a esta modalidad de control de uso de los datos es utilizar computación auditada. En este caso, las operaciones sobre los datos son permitidas pero también auditadas.

Las contribuciones de esta tesis son dos. En primer lugar, presentamos un sistema de tipos para Bytecode JVM que garantiza que el flujo de la información sea seguro, incluso en el caso en que diferentes instancias de una clase puedan tener atributos cuyo nivel de seguridad varía de acuerdo al contexto en el cual han sido instanciados. La seguridad viene dada por la propiedad de no-interferencia: una formulación rigurosa de la ausencia de filtraciones de información sensible. Si bien esta propiedad es técnicamente viable, en la práctica hay muchos programas que permiten revelar intencionalmente cierta información sensible. Esto nos motiva a extender el Bytecode JVM con un mecanismo de "desclasificación" que permite liberar información sensible. El riesgo que introduce es que el mismo pueda ser abusado, permitiendo desclasificar más información sensible de la que originalmente se tenía prevista. Por tal motivo, extendemos el sistema de tipos para garantizar que estas situaciones no sucedan. Dicha extensión satisface "desclasificación robusta": un atacante no puede deducir ninguna información adicional a la permitida por la política de seguridad ni tampoco puede influir en los datos que serán desclasificados.

En segundo lugar, presentamos un análisis teórico de un modelo computacional funcional que mantiene un registro de su historia de computación. A partir de un fragmento de la Lógica de Justificación (JL) de Artemov y el isomorfismo de Curry-de Bruijn-Howard obtenemos un cálculo lambda que modela unidades auditadas de computación. Reducción en estas unidades auditadas genera trazas que están confinadas a cada unidad auditada. Asimismo, estas unidades auditadas pueden consultar sus trazas para tomar decisiones en base a las mismas. Probamos seguridad de tipado y normalización fuerte para una restricción del mismo. Este cálculo lambda puede ser utilizado para definir programas que refuerzan una variedad de políticas de seguridad basadas en computación auditada.

Palabras Clave:

Flujo de Información, Desclasificación, No Interferencia, Robustes, Bytecode, Lógica de Justificación, Computaciones Auditadas, Isomorfismo de Curry-Howard.

Types for Security based-on Information Flow and Audited Computation

Abstract

One of the pillars of computer security is to ensure the confidentiality of information restricting its access to authorized users. Information Flow Analysis studies how information is transmitted by different mechanisms or channels. An attacker can obtain confidential information by observing the output (of any particular channel) of the system.

However, while information flow policies are useful and important, there are other proposed mechanisms such as access control or stack inspection. We want a programming language that can enforce a wide range of policies while providing the assurance that programs enforce their policies correctly. An alternative is to use audited trails (logs). Informally, a program is auditable if, at any audit point, an impartial judge is satisfied with the evidence produced by the program.

The contributions of this dissertation are twofold. Firstly, we present a type system for ensuring secure information flow in a JVM-like language that allows instances of a class to have fields with security levels depending on the context in which they are instantiated. We prove that the type system enforces noninterference.

Although noninterference is an appealing formalisation for the absence of leaking of sensitive information, its applicability in practice is somewhat limited given that many systems require intensional release of secret information. This motivates our extension of a core `JVM`-like language with a mechanism for performing downgrading of confidential information. We discuss how the downgrading mechanism may be abused in our language and then extend our type system for ensuring it captures these situations. It is proved that the type system enforces robustness of the declassification mechanism: attackers may not affect what information is released or whether information is released at all.

Secondly, we present a proof theoretical analysis of a functional computation model that keeps track of its computation history. A Curry-de Bruijn-Howard isomorphism of an affine fragment of Artemov's Justification Logic yields a lambda calculus $\lambda^{\natural}$ which models audited units of computation. Reduction in these units generates audit trails that are confined within them. Moreover, these units may look-up these trails and make decisions based on them. We affirm that the computational interpretation of **JL** is a programming language that records its computation history. We prove type safety for $\lambda^{\natural}$ and strong normalisation for a restriction of it. $\lambda^{\natural}$ can be used for writing programs that enforce a variety of security policies based on audited trails.

Keywords:

Information Flow, Declassification, Noninterference, Robustness, Bytecode, Justification Logic, Audited Computations, Curry-Howard Isomorphism.

A Yani, Guille y Benja
con todo mi amor.

Agradecimientos

La culminación de esta tesis se logro gracias al esfuerzo, el apoyo y los aportes (aportes no necesariamente relacionados con la tesis) de muchas personas y algunas instituciones. Quiero agradecer sinceramente a todos ellos:

Primero, a quienes más quiero y amo:

A Yani por su apoyo incondicional, por darme su amor y querer compartir su vida conmigo. Simplemente, te amo!!!

A Guille y Benja por su capacidad infinita de amar y de hacer travesuras.

A mis viejos, gracias por todo! Sin dudas, mi viejo me contagió el amor por la docencia universitaria.

A mis hermanos porque siempre están.

A mis amigos de toda la vida: Tincho, Boti, Lucas, Corcho, Nari, Mencho, Nico, Guille y Cali, por los buenos momentos pasados y los que quedan por pasar.

A mis amigos del DC-UNRC, Germán, Chino, Valen, Naza, Sonia, Marta, Gaston, Damian, Pablo, Ivana, Marcelo, Zurdo, Marcelo y Jorge.

A todos gracias por su cariño, comprensión y el tiempo compartido.

Gracias Eduardo! Por la guía, paciencia y enseñanzas brindadas. Sin lugar a dudas, el gran artífice de las cosas buenas de este trabajo.

El apoyo incondicional y la colaboración de dos grandes amigos, fue sin lugar a dudas lo que me permitió culminar con este trabajo. Gracias Tincho y Germán!!! Por estar siempre dispuestos a brindarme su amistad y ayuda.

Al Profe porque es un ejemplo a seguir. Gracias por las oportunidades brindadas y las enseñanzas dadas.

A Pedro, Diego y Alejandro, los jurados, por aceptar ser los revisores de este trabajo, la atenta lectura, las correcciones, críticas y los valiosos aportes que realizaron.

Un agradecimiento especial para Gilles, por estar siempre dispuesto a colaborar y brindar sus consejos.

Un especial agradecimiento a todo el Departamento de Computación de la FCEFQyN de la UNRC, un muy buen lugar de trabajo.

Al Departamento de Computación de la Facultad de Exactas de la UBA, en especial a la Subcomisión de Postgrado, a la Secretaría y a los profes de los cursos que tome (Alejandro, Diego, Juan Pablo y José). A Charlie, Juan Pablo y Mariano que compartieron su oficina cada vez que visitaba el DC.

Quiero agradecer al LIFIA, ITBA, FaMaF, UNQui e InCo instituciones que visite en el transcurso de mis estudios. Por último, al CONICET, el MinCyT de la Provincia de Córdoba, la UNRC, y la ANPCyT que me brindaron recursos para poder realizar mis estudios.

A todos ellos y a aquellos que me olvido en este momento:

Un abrazo y ... Gracias Totales!!!

Contents

List of Figures

1 Introduction

Nowadays, the *information age* has its own Pandora's box. In its large inventory, mobile code is one of the leading distributors of adversities. Mobile code is a powerful method to install and execute code dynamically. However, it represents a serious security risk for the receiver (for instance providing access to its resources and internal services). Therefore, this method can be used for malicious purposes, depending on the intentions of its creator or any interceptor.

Computer security has always been an important concern; the use of mobile code makes it even more relevant. One of the pillars of computer security is to ensure the confidentiality of private information maintaining its access only to authorized users. Data confidentiality can be impaired while transmitting from one system to another. This problem can be addressed using cryptographic techniques, however, an attacker could gain knowledge of the secret data by extracting them from the system. Confidentiality of secret stored data has been an active research field [53, 54, 117, 116, 82, 99, 19, 72].

To avoid leaking information, data can be encrypted. However, encryption is not applied due to efficiency reasons since it is expensive to make computing with encrypted data. Although it is possible to transmit confidential information on encrypted channels, it is desirable to provide security guarantee regarding how data is manipuled once decrypted. To access secret data, the attacker can apply two main approaches: (1) tampering the authentication mechanism, and (2) deducing information. In the first approach, the attacker may be able to access the information system with the identity and permissions of someone having access to secret data. The second approach deduces the secret information observing outputs of a system executing a precise process, malicious or not. The main techniques to prevent information leakage for these attacks are *access control* and *information flow analysis*.

Access control [36] is a technique to control who can interact with a resource. One of the limitations of this technique is that once access has been granted, there is no real control of the dissemination of the data. For instance, a process may need to access confidential data in order to fulfill its goal. An owner may agree to allow this process to access confidential data for the fulfillment of the process purpose. However, it is likely that the owner does not want the process to spread the information it has been given access to. Unfortunately, this spreading of information often can not be practically and efficiently enforced using access control. For example, a tax computation service needs access to the financial information of the client. It also needs access to the service provider server, for example, to send a bill if the service cost is on a per use basis or simply to download the latest updates. Then, what are the customers guarantees that its financial data are not unduly sent to the service provider? To enforce such a policy, it is required to enforce some sort of information flow control.

Information Flow Analysis (IFA) [99] studies how information is transmitted directly or indirectly by different mechanisms or channels. An attacker can obtain confidential information by observing the output of the system. The application of this technique can help to control propagation of confidential data. This is the reason why information flow control mechanisms are of high interest.

Research in language-based security has focused on how to ensure that information flows, within programs, do not violate the intended confidentiality properties. Language-based security is becoming an increasingly important aspect of computer security. Large amounts of data are being processed by computer programs and there is a need for control and accountability in the way these programs handle the information they glean from the data. The programming language community has focussed on information flow control: making sure that sharing of data is controlled in some manner to ensure that it does not leak in undesirable ways. Language technology, such as type systems, can be fruitfully applied to address this problem.

Language-based security is a useful technique that can be applied in many situations. It can range from a simple syntactic parsing to a full semantic analysis, such as verifying code security properties, finding potential bugs in applications, detecting unused code, applying reverse engineer code, to name a few. These techniques can be applied to any programming language (with varying challenges depending of the source language). In the case of Java, they can be applied to the source code or the bytecode code. Applying these techniques to bytecode increases their applicability since the source code for some programs might not be available.

Currently a large amount of code is distributed over the Internet largely in the form of bytecode or similar low-level languages. Direct verification of security properties of bytecode allows the code consumer to be assured that its security policy is upheld. Analysis at the bytecode level has the benefit of being independent of a mechanism (compiled, handwritten, code generated) that generated it.

Although initial work was developed for imperative languages, most current efforts are geared towards object-oriented and concurrent languages. Somewhat orthogonally, and sparked mainly by the success of Java and the JVM [75], also low-level code formats are being studied. This allows the direct analysis of Java applets and other code distributed over the Internet. A number of systems in the literature have focused on type based IFA for bytecode (cf. Section 3.7.2). Advanced features including object creation, exceptions, method calls, etc. [116, 82, 84, 19, 108, 105, 94] have been considered. However, many of these systems are crippled in several important ways that reduce its potential applicability, already for a core fragment of the language.

The first part of this thesis is concerned with language-based information flow [99] policies for mobile code. We propose a static enforcement mechanism of confidentiality policies for low-level languages that allows verification using type system for Java Bytecode.

While information flow policies are useful and important, there are many other styles of policies that are in common use, including access control [36], type enforcement [14], tainting [43], stack inspection [55], and forms of security automata [121]. One approach to verifying the correct enforcement of these policies is to encode them on programs. Then, we want a programming language that can enforce a wide range of policies while providing the assurance that programs enforce their policies correctly. An alternative is to use audited trails (logs).

Auditing is an essential component of secure distributed applications, and the use of audit logs is strongly recommended by security standards [1, 2]. Informally, a program is auditable if, at any audit point, an impartial judge is satisfied with the evidence produced by the program. In practice, most applications selectively store information of why authorizations were granted or why services were provided in audit logs, with the hope that this data can be later used for regular maintenance, such as debugging of security policies, as well as conflict resolution.

As a step toward this goal, in second part of those thesis, we present a proof theoretical analysis of a λ-calculus which produces a trail of its execution (as a type system for a calculus that records its computation history). This λ-calculus can be used for writing programs that enforce a variety of security policies based on audited trails. The programmers may specify security policies in a separate part of the program called the enforcement policy.

1.1 Structure of the Thesis and Contributions

This thesis is organized in two parts. The first part is concerned with practical, language-based information flow policies for mobile code. We address the issue of what is a secure bytecode program from the point of view of confidentiality in information flow. We propose a static enforcement mechanism for low-level languages to confidentiality policies that allows verification using a type system for Java Bytecode.

Static, type-based information flow analysis techniques targeted at Java and JVM-like code typically assume a global security policy on object fields: all fields are assigned a fixed security level. However, instances of objects may be created under varying security contexts, particularly for widely used classes such as wrapper or collection classes. This entails an important loss in precision of the analysis.

In ***Chapter 3***, we present a core bytecode language together with a sound type system that performs information flow analysis in a setting where the security level assigned to fields may vary at different points of the program where their host object is created. Two further contributions, less significant although worthy of mention, are that local variables are allowed to be reused with different security levels (flow-sensitive type system) and also that we enhance precision on how information flow through the stack is handled.

Chapter 3 is an extended version of the paper [28] presented at the 23rd Annual ACM Symposium on Applied Computing (SAC'08), Fortaleza, Brazil, March 16-20, 2008.

The noninterference property states that a system is void of insecure information flows from private to public data. This constitutes too severe a requirement for many systems which allow some form of controlled information release. Therefore, an important effort is being invested in studying forms of intended information declassification. Robust declassification is a property of systems with declassification primitives that ensures that a third party cannot affect the behavior of a system in such a way as to declassify information other than intended.

In ***Chapter 4***, we study robust declassification for a core fragment of Java Bytecode. We observe that, contrary to what happens in the setting of high-level languages, variable reuse, jumps and operand stack manipulation can be exploited by an attacker to declassify information that was not intended to be released. We introduce a sound type system for ensuring robust declassification.

Chapter 4 is an extended version of the paper [29] presented at the V Congreso Iberoamericano de Seguridad Informática (CIBSI'09), Montevideo, Uruguay, November 16-18, 2009.

The second part of this thesis presents a proof theoretical analysis of a λ-calculus which produces a trail of its execution. Justification Logic (**JL**) [9, 10, 11] is a refinement of modal logic that has recently been proposed for explaining well-known paradoxes arising in the formalization of Epistemic Logic. Assertions of knowledge and belief are accompanied by justifications: the formula $[\![t]\!]A$ states that t is "reason" for knowing/believing A.

In ***Chapter 5***, we study the computational interpretation of **JL** via the Curry-de Bruijn-Howard isomorphism in which the modality $[\![t]\!]A$ is interpreted as: t is a type derivation justifying the validity of A. The resulting lambda calculus is such that its terms are aware of the reduction sequence that gave rise to them. This serves as a basis for understanding systems, many of which belong to the security domain, in which computation is history-aware.

Chapter 5 is an extended version of the paper [30] presented at the 7th International Colloquium on Theoretical Aspects of Computing (ICTAC 2010), Natal, Brazil, September 1-3, 2010.

The main definitions and statements of the results are presented in the body of the thesis. Proofs are deferred to the appendices.

Part I

Information Flow for Bytecode

2

Preliminaries

We introduce some basic concepts that serve as the basis for the developments presented in this part of the thesis.

Structure. Section 2.1 presents, informally, concepts related to information flow, noninterference and declassification. The Java Virtual Machine and the Java Bytecode are presented in Section 2.2.

2.1 Information Flow

Some work in *Language-Based Security* [70, 104] approach consists of preserving the relevant information obtained from the source code into its compiled version. The extra information, called the certificate, is obtained during the compilation process and is included in the compiled code. The user can then carry out an analysis of the code as well as its certificate to check that it fits the security policy requirements. If the certificate proves to be secure then the code can be safely executed. The main advantage of this procedure lies in the fact that the code producer must assume the costs of guaranteeing the code security (by generating the certificate) whereas the user has only to verify whether this certificate fits the security requirements [70]. Among the *Language-Based Security* variants techniques to guarantee security we can mention: *Proof-Carrying Code* (PCC) [89], *Type Assembly Language* (TAL) [80] and *Efficient Code Certification* (ECC) [70]. All these techniques introduces logic frameworks and type systems that guarantee security; and they all use the information generated during the compilation process so that the consumer can be able to verify the code security efficiently. However, they differ in expressiveness, flexibility and efficiency.

A program is *information flow secure* if an attacker cannot obtain confidential information by interacting with the program. The starting point in secure information flow analysis is the classification of program variables into different confidential security levels. Following standard practice, we assume security levels to form a lattice. However, in most of our examples, we will only use two security levels: public information (*low* or `L`) and secret information (*high* or `H`). Security levels are attributed to variables and signals, using subscripts to specify them (eg. $x_{\mathtt{H}}$ is a variable of high level). In a sequential imperative language, an insecure flow of information occurs when the initial values of high variables influence the final value of low variables. For example, if $\mathtt{L} \leq \mathtt{H}$, then we would allow flows from `L` to `L`, from `H` to `H`, and from `L` to `H`, but we would disallow flows from `H` to `L` [106].

The simplest case of insecure flow, called *explicit* (insecure) flow, is the assignment of the value of a high variable to a low variable, as in

$$y_{\mathtt{L}} = x_{\mathtt{H}}$$

More subtle kinds of flow, called *implicit* flows, may be induced by the control flow. An example is the program

$$\mathtt{if}\quad x_{\mathtt{H}} = 0 \quad \mathtt{then} \quad y_{\mathtt{L}} = 0 \quad \mathtt{else} \quad y_{\mathtt{L}} = 1$$

where at the end of execution the value of $y_{\mathtt{L}}$ may give information about $x_{\mathtt{H}}$.

One of the difficulties of tracking information flows is that information may flow through different communications channels. To address this issue, Lampson [71] introduces the concept of covert channels. Various convert channels have been studied, such as, timing [120, 7, 107, 100, 96, 97], termination [119, 4] and power consumption [69]. Gligor, Millen et al. [61] give a comprehensive introduction of covert channels for both theoretical and practical purposes.

Another property related to information flow is *integrity*. Integrity and confidentiality are long considered dual to each other [34]. Confidentiality prevents data flow from flowing to undesired destinations. Integrity prevents critical computations from being manipulated from outside. There is an important amount of research enforcing integrity properties using information flow analysis [83, 119, 64, 85, 19, 105, 74].

In general, two different approaches have been explored with the aim of providing information flow, static and dynamic, each associated with compile-time and run-time systems. Run-time solutions take a different approach by using the labels as an extra property of the object and tracking their propagation as the objects are involved in computation. In the compile-time approach, applications are written in specially designed programming languages in which special annotations are used to attach security labels and constraints to the objects in the program. At compile time, the compiler uses these extra labels to ensure authorized flows of information. These compile-time checks can be viewed as an augmentation of type checking [87, 86]. In type-based analysis, types are extended so that they include security information. Then typing rules are constructed by taking security information into account and, finally, a type checker algorithm for the type system is developed.

In the last years, type-based information flow analysis has been drawing much attention. Many information flow analysis techniques based on type systems have been proposed for various kinds of languages, including low-level languages [68, 25, 94, 58, 28, 38, 124], procedural [116, 117], functional [105, 64], object-oriented [17, 109, 82] and concurrent languages [107, 120, 41, 97].

2.1.1 Noninterference

Following Cohen's notion of strong dependency [45], Goguen and Meseguer [62] introduced the general concept of *noninterference* to formalize security policies. Volpano, Smith, and Irvine [117] are the first in formalizing the definition of noninterference for information flow.

Noninterference is a correctness property of information flow analysis. This property intuitively means that, given a program that takes a confidential input and yields a public output, the public output remains the same no matter what value is given as the confidential input. I.e. in a system with multiple security levels, information should only be allowed to flow from lower to higher (more secure) levels [62]. Much research in the area of secure information flow formalize the notion of secure information flow based on noninterference.

2.1.2 Flow-sensitivity vs. flow-insensitivity

Enforcing noninterference requires the information flow analysis throughout the control flow of the program. The control flow of a program plays an important role in static analysis as well as in its precision. With respect to the interpretation of control flow, program techniques can be divided into two categories: flow-insensitive and flow-sensitive.

Flow-insensitive analysis is independent of the control flow encountered as it does not consider the execution order. The program is considered as a set of statements. Information given simply indicates that a particular fact may hold anywhere in the program because it does hold somewhere in the program. Flow-sensitive analysis [65] depends on control flow. The program is considered as a sequence of statements. A given piece of information indicates that a particular fact is true at a certain point in the program. Considering that $x_{\mathtt{L}}$ is a public, low variable and $y_{\mathtt{H}}$ is a secret, high variable, the example

$$x_{\mathtt{L}} = y_{\mathtt{H}};$$
$$x_{\mathtt{L}} = 0;$$

yields an illegal flow of information in a flow-insensitive analysis (but not in the flow-sensitive analysis), yet the program clearly satisfies noninterference. Hence, flow-insensitive information is fast to compute, but not very precise. Flow-sensitive analysis usually provides more precise information than flow-insensitive analysis but it is also usually considerably more expensive in terms of computational time and space. Flow-sensitive approach is important in presence of register reuse, a significant feature in low-level languages.

2.1.3 Declassification

Noninterference is a good baseline for guaranteeing that there is no secret information leakage. However, noninterference is of limited use in practice since many systems require intensional release of secret information. Indeed, full noninterference is not always desired [126] in real applications because in many cases information downgrading is part of the desired policy. Examples are: releasing the average salary from a secret database of salaries for statistical purposes, a password checking program, releasing an encrypted secret, etc. This calls for relaxed notions of noninterference where primitives for information declassification are adopted. Yet, declassification is hard to analyze because it breaks noninterference. A declassification mechanism can be exploited affecting the data that is declassified or whether declassification happens at all. There are many ways to define the semantics of declassification properly.

A recent survey on declassification [101, 102] categorizes approaches to information release by *what* information is declassified, by *whom* it is released, and *where* and *when* in the system declassification occurs. Robustness operates primarily on the *who* dimension since it controls that the attacker does not decide when data get declassified.

In this same survey on declassification [101, 102], the authors identify some common semantic principles for declassification mechanisms:

1. *semantic consistency*, which states that security definitions should be invariant under equivalence-preserving transformations;
2. *conservativity*, which states that the definition of security should be a weakening of noninterference;
3. *monotonicity* of release, which states that adding declassification annotations cannot make a secure program become insecure. Roughly speaking: the more you choose to declassify, the weaker the security guarantee; and
4. *non-occlusion*, which states that the presence of declassifications cannot mask other covert information leaks.

The authors state that these principles help shed light on existing approaches and should also serve as useful *sanity check* for emerging models. For instance, they claim that robust declassification satisfies these principles.

2.2 The Java Virtual Machine

The Java programming language [63] is a general-purpose object-oriented language. Java is currently one of the most popular programming languages in use, and is widely used from application software to web applications [3]. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Furthermore, it omits many of the features that make C and C++ complex, confusing, and unsafe. The Java platform was initially developed to address the problems of building software for networked consumer devices. It was designed to support multiple host architectures and to allow secure delivery of software components. To meet these requirements, compiled code had to survive transport across networks, operate on any client, and assure the client that it was safe to run.

Java is specifically designed to have as few implementation dependencies as possible, which means that computer programs written in the Java language must run similarly on any supported hardware/operating-system platform. This is achieved by compiling the Java language code to an intermediate representation called Java bytecode, instead of directly to platform-specific machine code. Java bytecode instructions are analogous to machine code, but are intended to be interpreted by a (Java) Virtual Machine (JVM) [75] written specifically for the host hardware.

The JVM is the cornerstone of the Java. It is the component responsible for its hardware and operating system independence, the small size of its compiled code, and its ability to protect users from malicious programs.

The JVM knows nothing of the Java programming language, only of a particular binary format, the class file format. A class file contains JVM instructions (or bytecodes) and a symbol table, as well as other ancillary information. A JVM can be also used to implement programming languages other than Java. For example, Ada [6] or Scala [103] source code can be compiled to Java bytecode to then be executed by a JVM.

A basic philosophy of Java is that it is inherently *safe* from the standpoint that no user program can *crash* the host machine or otherwise interfere inappropriately with other operations on the host machine, and that it is possible to protect certain functions and data structures belonging to *trusted* code from access or corruption by *untrusted* code executing within the same JVM. Furthermore, common programmer errors that often lead to data corruption or unpredictable behavior such as accessing off the end of an array or using an uninitialized pointer are not allowed to occur. Several features of Java combine to provide this safety, including the *class model*, the *garbage-collected heap*, and the *Bytecode Verifier*.

The Bytecode Verifier *verifies* all bytecode before it is executed [75]. This verification consists primarily of the following checks:

- There are no operand stack overflows or underflows.
- All local variable uses and stores are valid.
- The arguments to all the Java virtual machine instructions are of valid types.
- Branches are always to valid locations.
- Data is always initialized and references are always type-safe.
- Any given instruction operates on a fixed stack location.
- Methods are invoked with the appropriate arguments.
- Fields are assigned only using values of appropriate types.
- Arbitrary bit patterns cannot get used as an address.
- Access to *private* data and methods is rigidly controlled.

The Bytecode Verifier is independent of any compiler. It should certify all code generated by any compiler. Any class file that satisfies the structural criteria and static constraints will be certified by the verifier. The class file verifier is also independent of the Java programming language. Programs written in other languages can be compiled into the class file format, but will pass verification only if all the same constraints are satisfied [75].

Java bytecode language consists of 212 instructions, then is hard to perform its formalisation. For this reason, the existing formalisations usually cover a representative set of instructions. Chrzaszcz, Czarnik and Schubert [44] describe a concise formalisation of JVM bytecode which turns out to be factorisable into 12 instruction mnemonics. It reduces the number of instructions in a systematic and rigorous way into a manageable set of more general operations that cover the full functionality of the Java bytecode. The factorization of the instruction set is based on the use of the runtime structures such as operand stack, heap, etc. We consider 11 of these instruction mnemonics in the development of this work.

The instruction set can be roughly grouped as follows:

Stack operations: Constants can be pushed onto the stack, e.g., `iconst_0` or `bipush` (push byte value).
Arithmetic operations: The instruction set of the Java Virtual Machine distinguishes its operand types using different instructions to operate on values of specific type. Arithmetic operations starting with i, for example, denote an integer operation. E.g., `iadd` that adds two integers and pushes the result back on the stack. The Java types boolean, byte, short, and char are handled as integers by the JVM.
Control flow: There are branch instructions like `goto`, and `if_icmpeq`, which compares two integers for equality. Exceptions may be thrown with the `athrow` instruction. Branch targets are coded as offsets from the current byte code position, i.e., with an integer number.
Load and store operations for local variables: like `iload` and `istore`. There are also array operations like `iastore` which stores an integer value into an array.
Field access: The value of an instance field may be retrieved with `getfield` and written with `putfield`.
Method invocation: Static Methods may either be called via `invokestatic` or be bound virtually with the `invokevirtual` instruction.
Object allocation: Class instances are allocated with the `new` instruction, arrays of basic type like `int[]` with `newarray`.
Conversion and type checking: For stack operands of basic type, there exist casting operations like `f2i` which converts a float value into an integer. The validity of a type cast may be checked with `checkcast` and the `instanceof` operator can be directly mapped to the equally named instruction.

3

Type-Based Information Flow Analysis for Bytecode Languages with Variable Object Field Policies

We present JVM^s ("s" is for "safe"), a core bytecode language together with a type system that performs IFA in a setting where the security level assigned to fields may vary at different points of the program where their host object is created. Two further contributions are that local variables are allowed to be reused with different security levels and also that we enhance precision on how information flow through the stack is handled.

Structure. Section 3.1 introduces motivating examples and Section 3.2 presents the syntax and operational semantics of JVM^s. The type system is presented in Section 3.3. Exceptions are added in Section 3.4 and method calls in Section 3.5, followed by an analysis of noninterference in Section 3.6. Finally, further language features and related work are briefly discussed in Section 3.7.

3.1 Motivation

Most existing literature on type-based IFA [16, 19, 25, 26, 24] assumes that the fields of objects are assigned some fixed security label. However, some objects such as instances of, say, class Collection or WrapInt (wrapper class for integers), are meant to be used in different contexts. Consider the aforementioned WrapInt class that has a field for storing integers. If the field is declared public, then wrappers that are instantiated in high-level regions (fragments of code that depends on confidential data) cannot have their integers assigned. Likewise, if it is declared secret, then although it may be freely assigned in both low and high-level regions, precision is lost when it is used in low-level regions (fragments of code that does not depend on confidential data). This is illustrated by the Example 3.1. This program loads the value of $x_{\mathtt{H}}$, which we assume to be secret, on the stack, creates a new instance of WrapInt and assigns 0 to its only field f (the possible security levels for f and their consequences are discussed below). It then branches on the secret value of $x_{\mathtt{H}}$. Instructions 6 to 8 thus depend on this value and hence the value 1 assigned to field f of the new instance of WrapInt shall be considered secret.

Example 3.1.

```
1 load      x_H
2 new       WrapInt
3 push      0
4 putfield  f
5 if        9
6 new       WrapInt
7 push      1
8 putfield  f
9 return
```

If field f of WrapInt is declared public, then the assignment on line 4 is safe but not the one on line 7. Since this would render the wrapper useless, one could decide to declare f secret. This time both assignments are acceptable, however the public value 0 is unnecessarily coerced to secret with the consequent loss in precision.

Local variable reuse. Local variable reuse is promoted by Sun's JVM Specification [75, Ch.7]: "*The specialized load and store instructions should encourage the compiler writer to reuse local variables as much as is feasible. The resulting code is faster, more compact, and uses less space in the frame*". Java Card [110], the reduced Java language for smart cards, also encourage this form of reuse particularly due to the limited available resources. It also promoted object reuse: reusing attributte of objects. This form of reuse reduces the wear of the smart card's persistent memory and is even necessary in most cases given that garbage collection is not readily available. Finally, reuse is of key importance in other low-level languages such as assembly language and although we do not address them here (cf. [80, 38]) the techniques developed in this work should be applicable.

The following code excerpt, in Example 3.2, illustrates variable reuse. Assuming that x is declared public and y secret, line 6 attempts to reuse x to store a secret value.

Example 3.2.

1 `push` 1
2 `store` $x_{\mathtt{L}}$ $\leftarrow$ *x is public*
3 `load` $y_{\mathtt{H}}$
4 `if` 7
5 `new` C
6 `store` $x_{\mathtt{H}}$ $\leftarrow$ *x is secret*
7 $\cdots$

Insensitive-flow type-based IFA for high-level programming languages reject this code on the grounds that confidentiality could be breached: a public variable is assigned a secret value. However, this is perfectly secure code given that x now takes on a new value (and security level) unrelated to the previous value (and use) of x and indeed no security risk is present.

Likewise, Example 3.3 presents no information leak given that the second use of x on line 6 completely discards (and is unrelated to) the first one. In the following example, we assume both x and y are declared secret whereas z is declared public. Most extant type systems [24, 94, 118] will in fact complain that illicit information flow takes place at line 8.

Example 3.3.

1 `load` $y_{\mathtt{H}}$
2 `if` 5
3 `new` C
4 `store` $x_{\mathtt{H}}$
5 `push` 0
6 `store` $x_{\mathtt{L}}$
7 `load` $x_{\mathtt{L}}$
8 `store` $z_{\mathtt{L}}$
9 $\cdots$

Even though variable reuse can be avoided in the analysis by transforming the bytecode so that different uses of a variable become different variables [73], this makes the security framework dependent on yet another component. Moreover, this transformation is not viable in other low-level code such as assembly language where the number of registers is fixed.

Operand stack limitations. The operand stack introduces new ways in which information may leak, these being notably absent in high-level languages given that the stack is not directly available to the user. A number of articles take the approach of modifying the level of the elements of the stack when evaluating conditionals [25, 26, 24]. In particular, whenever a conditional on a secret expression is evaluated, the security elements of *all* the elements in the stack are set to secret. That certainly helps avoid undesired leaks but, arguably, in a rather drastic way. In particular, once the dependence region of the conditional exits, all remaining values on the operand stack shall be tagged as secret limiting the expressiveness of the analysis.

It is often argued that since Java compilers use the operand stack for evaluation of expressions and thus is empty at the beginning and end of each instruction, the aforementioned approach is not a severe limitation (the compilation of the expression $e1?e2 : e3$ does, however, require dealing with nonempty operand stacks). However, this makes the IFA on low-level code unnecessarily dependent, as shown in this work, on assumptions on the compiler or the application that generated the bytecode. In turn, this requires studying IFA of the low-level object code itself, the topic of this work. Such an analysis should ideally strive to be applicable independently of how the code was generated hence reducing the trusted base of the security framework.

As an example of the loss in precision, using the approach of modifying the level of the elements of the stack when evaluating conditionals, consider the piece of code in Example 3.4. We assume x and y are declared secret and z public. This program fails to type check [25, 26, 24] at line 9. An attempt is made to store the topmost element of the stack (0, pushed on the stack in line 1) into z. Although this constant is public and was added to the stack as such, after exiting the dependence region of the conditional at line 3, its level was changed to secret. Hence, it can no longer be assigned to the public variable z.

Example 3.4.

```
1  push   0
2  load   x_H
3  if   7
4  push   0
5  store   y_H
6  goto   9
7  push   0
8  store   y_H
9  store   z_L
10 ···
```

A further often seen limitation [38] is that public elements on the stack are not allowed to be freed under high-level regions[1]. Although this avoids some situations of illicit flows it is overly restrictive. In particular, Example 3.5 is perfectly safe: there is no risk in popping the low element stack 1 as long as it is popped in *both* branches.

Example 3.5.

```
1 push   1
2 load   x_H
3 if   6
4 pop
5 goto   8
6 pop
7 goto   8
8 return
```

[1] This particular problem is not present in those works which update the level of all elements on the stack. However, this is because all elements in the stack are promoted to high security, as already mentioned. This has the consequent loss in precision discussed in the previous item.

3.2 Syntax and Semantics

A *program* B is a sequence of bytecode instructions as defined in Figure 3.1. There *op* is $+$ or $\times$; x ranges over a set of *local variables* $\mathbb{X}$; f, g ranges over a fixed set of *field identifiers* $\mathbb{F}$; C ranges over a universe $\mathbb{C}$ of *class names*; i over the natural numbers $\mathbb{N}$; j over the integers $\mathbb{Z}$.

I ::=	`prim` *op*	primitive operation
|	`push` j	push i on stack
|	`pop`	pop from stack
|	`load` x	load value of x on stack
|	`store` x	store top of stack in x
|	`if` i	conditional jump
|	`goto` i	unconditional jump
|	`return`	return
|	`new` C	create new object in heap
|	`getfield` f	load value of field f on stack
|	`putfield` f	store top of stack in field f

Fig. 3.1. Bytecode instructions

We write $\mathsf{Dom}(B)$ for the set of *program points* of B and $B(i)$, with $i \in 1..n$ and n the length of B, for the i^{th} instruction of B.

Example 3.6. For example, in

1 `load` $x_{\mathtt{H}}$
2 `new` WrapInt
3 `push` 0
4 `putfield` f
5 `if` 9
6 `new` WrapInt
7 `push` 1
8 `putfield` f
9 `return`

we have $\mathsf{Dom}(B) = \{1, 2, .., 9\}$, $B(1) =$ `load` $x_{\mathtt{H}}$, $B(2) =$ `new` WrapInt and so forth.

A *value* v is either an integer j, a (heap) location o or the null object *null*. Thus, if $\mathbb{L}$ stands for the set of locations, $\mathbb{V} = \mathbb{Z} + \mathbb{L} + \{null\}$. *Objects* are modeled as functions assigning values to their fields[2], $\mathbb{O} = \mathbb{C} \times (\mathbb{F} \to \mathbb{V})$.

Machine states are tuples $\langle i, \alpha, \sigma, \eta \rangle$, where $i \in \mathbb{N}$ is the program counter that points to the next instruction to be executed; α (local variable array) is a mapping from local variables to values; σ (stack) is an operand stack, the operand stacks are list of values; and η (heap) is a mapping from locations to objects. A machine state *for* B is one in which the program counter points to an element in $\mathsf{Dom}(B)$. These definitions are summarized in Figure 3.2.

The small-step operational semantics of JVM^s is standard; it is defined in terms of the one-step reduction schemes given in Figure 3.3. For example, the scheme R-Store resorts to the following notation: If $\alpha \in \mathbb{LV}$, $x \in \mathbb{X}$ and $v \in \mathbb{V}$, then $\alpha \oplus \{x \mapsto v\}$ is the local variable array α' s.t. $\alpha'(y) = \alpha(y)$ if $y \neq x$,

[2] We assume that the field names of all classes are different.

$\mathbb{V}$	$= \mathbb{Z} + \mathbb{L} + \{null\}$	Values
$\mathbb{O}$	$= \mathbb{C} \times (\mathbb{F} \to \mathbb{V})$	Objects
$\mathbb{LV}$	$= \mathbb{X} \to \mathbb{V}$	Local Variables
$\mathbb{S}$	$= \mathbb{V}^*$	Operand Stack
$\mathbb{H}$	$= \mathbb{L} \to \mathbb{O}$	Heap
$State$	$= \mathbb{N} \times \mathbb{LV} \times \mathbb{S} \times \mathbb{H}$	States
$FinalState$	$= \mathbb{V} \times \mathbb{H}$	Final States

Fig. 3.2. Memory Model

and $\alpha'(x) = v$ otherwise. For every function $f : A \to B$, $x \in A$ and $y \in B$, we denote with $f \oplus \{x \mapsto y\}$ the unique function f' such that $f'(y) = f(y)$ if $y \neq x$, and $f'(x) = y$ otherwise.

Execution of `new` relies on two auxiliary functions. One is Fresh that given a heap η creates a location o with $o \notin \mathsf{Dom}(\eta)$. The other is $\mathsf{Default}$ that given a class name returns an object of that class whose fields have been initialized with default values. The default values are zero for integer fields and *null* for object fields.

We write $s_1 \longrightarrow_B s_2$ if both s_1, s_2 are states for B and s_2 results by one step of reduction from s_1. In the sequel we work with states for some program B which shall be clear from the context and thus simply speak of states without explicit reference to B. Expressions of the form $\langle v, \eta\rangle$ are referred to as *final states.* A non-final state s may either reduce to another non-final state or to a final state. Initial states take the form $\langle 1, \alpha, \epsilon, \eta\rangle$, where ϵ denotes the empty stack. Summing up, $\longrightarrow_B \subseteq State \times (State + FinalState)$. We write $\twoheadrightarrow_B$ for the reflexive-transitive closure of $\longrightarrow_B$.

$$\text{(R-PUSH)}\quad \frac{B(i) = \texttt{push}\ j}{\langle i, \alpha, \sigma, \eta\rangle \longrightarrow_B \langle i+1, \alpha, j\cdot\sigma, \eta\rangle}
\qquad
\text{(R-POP)}\quad \frac{B(i) = \texttt{pop}}{\langle i, \alpha, v\cdot\sigma, \eta\rangle \longrightarrow_B \langle i+1, \alpha, \sigma, \eta\rangle}$$

$$\text{(R-LOAD)}\quad \frac{B(i) = \texttt{load}\ x}{\langle i, \alpha, \sigma, \eta\rangle \longrightarrow_B \langle i+1, \alpha, \alpha(x)\cdot\sigma, \eta\rangle}
\qquad
\text{(R-STORE)}\quad \frac{B(i) = \texttt{store}\ x}{\langle i, \alpha, v\cdot\sigma, \eta\rangle \longrightarrow_B \langle i+1, \alpha \oplus \{x \mapsto v\}, \sigma, \eta\rangle}$$

$$\text{(R-PRIM)}\quad \frac{B(i) = \texttt{prim}\ op}{\langle i, \alpha, n\cdot m\cdot\sigma, \eta\rangle \longrightarrow_B \langle i+1, \alpha, n\ op\ m\cdot\sigma, \eta\rangle}
\qquad
\text{(R-GOTO)}\quad \frac{B(i) = \texttt{goto}\ i'}{\langle i, \alpha, \sigma, \eta\rangle \longrightarrow_B \langle i', \alpha, \sigma, \eta\rangle}$$

$$\text{(R-IF1)}\quad \frac{B(i) = \texttt{if}\ i' \quad v \neq 0}{\langle i, \alpha, v\cdot\sigma, \eta\rangle \longrightarrow_B \langle i+1, \alpha, \sigma, \eta\rangle}
\qquad
\text{(R-NEW)}\quad \frac{B(i) = \texttt{new}\ C \quad o = \mathsf{Fresh}(\eta)}{\langle i, \alpha, \sigma, \eta\rangle \longrightarrow_B \langle i+1, \alpha, o\cdot\sigma, \eta \oplus \{o \mapsto \mathsf{Default}(C)\}\rangle}$$

$$\text{(R-IF2)}\quad \frac{B(i) = \texttt{if}\ i' \quad v = 0}{\langle i, \alpha, v\cdot\sigma, \eta\rangle \longrightarrow_B \langle i', \alpha, \sigma, \eta\rangle}
\qquad
\text{(R-PTFLD)}\quad \frac{B(i) = \texttt{putfield}\ f \quad o \in \mathsf{Dom}(\eta) \quad f \in \mathsf{Dom}(\eta(o))}{\langle i, \alpha, v\cdot o\cdot\sigma, \eta\rangle \longrightarrow_B \langle i+1, \alpha, \sigma, \eta \oplus \{o \mapsto \eta(o) \oplus \{f \mapsto v\}\}\rangle}$$

$$\text{(R-RET)}\quad \frac{B(i) = \texttt{return}}{\langle i, \alpha, v\cdot\sigma, \eta\rangle \longrightarrow_B \langle v, \eta\rangle}
\qquad
\text{(R-GTFLD)}\quad \frac{B(i) = \texttt{getfield}\ f \quad o \in \mathsf{Dom}(\eta) \quad \eta(o, f) = v}{\langle i, \alpha, o\cdot\sigma, \eta\rangle \longrightarrow_B \langle i+1, \alpha, v\cdot\sigma, \eta\rangle}$$

Fig. 3.3. Operational Semantics of JVM^s

l, l_i, λ	$\in \{\mathtt{L}, \mathtt{H}\}$	Security Levels
a, a_i	$\in$ *SymLoc*	Symbolic Locations
κ, κ_i	$\in$ *SecLabels*	Security Labels
κ	$::= \langle \{a_1, \ldots, a_n\}, l \rangle$	Security Label
ObjType	$::= [f_1 : \kappa_1, \ldots, f_n : \kappa_n]$	Object Types
V, V_i	$::= \mathbb{X} \to$ *SecLabels*	Frame Type
S, S_i	$::= \mathbb{N} \to$ *SecLabels*	Stack Type
T, T_i	$::=$ *SymLoc* $\to$ *ObjType*	Heap Type
$\mathcal{V}$	$= \{V_i \mid i \in 0..\mathsf{Dom}(B)\}$	Variable Typing Contexts
$\mathcal{S}$	$= \{S_i \mid i \in 0..\mathsf{Dom}(B)\}$	Stack Typing Contexts
$\mathcal{A}$	$= \{l_i \mid i \in 0..\mathsf{Dom}(B)\}$	Instruction Typing Contexts
$\mathcal{T}$	$= \{T_i \mid i \in 0..\mathsf{Dom}(B)\}$	Heap Typing Contexts

Fig. 3.4. Typing Contexts

3.3 Type System

We assume given a set $\{\mathtt{L}, \mathtt{H}\}$ of *security levels* (l) equipped with $\preceq$, the least partial order satisfying $\mathtt{L} \preceq \mathtt{H}$, and write $\sqcup, \sqcap$ for the induced supremum and infimum. *Security labels* (κ) are expressions of the form $\langle \{a_1, \ldots, a_n\}, l \rangle$ where a_i, $i \in 0..n$, ranges over a given infinite set *SymLoc* of *symbolic locations* (motivated shortly). We write *SecLabels* for the set of security labels. If $n = 0$, then the security label is $\langle \emptyset, l \rangle$. We occasionally write $\langle \_, l \rangle$, or l, when the set of symbolic locations is irrelevant. Furthermore, we note $\lfloor \langle R, l \rangle \rfloor = l$ and $\lceil \langle R, l \rangle \rceil = R$.

Two orderings on security labels are defined:

$$\langle R_1, l_1 \rangle \sqsubseteq \langle R_2, l_2 \rangle \text{ iff } R_1 \subseteq R_2 \text{ and } l_1 \preceq l_2$$

and

$$\langle R_1, l_1 \rangle \mathrel{\underset{\sim}{\sqsubset}} \langle R_2, l_2 \rangle \text{ iff } R_1 \subseteq R_2 \text{ and } l_1 = l_2$$

We write $\kappa \not\sqsubseteq \kappa'$ when $\kappa \sqsubseteq \kappa'$ does not hold. and $\kappa \mathrel{\not\underset{\sim}{\sqsubset}} \kappa'$ when $\kappa \mathrel{\underset{\sim}{\sqsubset}} \kappa'$ does not hold.

By abuse of notation, we write $\kappa \sqsubseteq l$ ($l \sqsubseteq \kappa$) to indicate that the level of κ (l) is below or equal to l (κ) and $\kappa \neq l$ to indicate that the level of κ is different from l.

The supremum on security labels over $\sqsubseteq$ is defined as follows:

$$\langle R_1, l_1 \rangle \sqcup \langle R_2, l_2 \rangle = \langle R_1 \cup R_2, l_1 \sqcup l_2 \rangle.$$

Also, we introduce the following abbreviations: $l_1 \sqcup \langle R_2, l_2 \rangle = \langle R_2, l_1 \sqcup l_2 \rangle$ and $\langle R_1, l_1 \rangle \sqcup l_2 = \langle R_1, l_1 \sqcup l_2 \rangle$.

Typing contexts supply information needed to type each instruction of a method. As such, they constitute families of either functions ($\mathcal{V}, \mathcal{S}, \mathcal{T}$) or labels ($\mathcal{A}$) indexed by a finite set of instruction addresses.

Definition 3.7 (Typing Context). *A* ***typing context for a program*** *B is a tuple $(\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T})$ where:*

- *$\mathcal{V}$ (variable typing) assigns a frame type V to each instruction of B. A frame type is a function assigning security labels to each local variable.*
- *$\mathcal{S}$ (stack typing) assigns a stack type S to each instruction of B. A stack type is a function assigning security labels to numbers from 0 to $n-1$, where n is assumed to be the length of the stack.*
- *$\mathcal{A}$ (program point typing) indicates the level ($\mathtt{L}$ or $\mathtt{H}$) of each instruction of B.*
- *$\mathcal{T}$ (heap typing), associates a heap type T to each instruction of B. Heap types map symbolic locations to object types, as explained below.*

We write $\mathcal{V}_i$ to refer to the i^{th} element of the family and similarly with the others typings. For example, $\mathcal{A}_i$ is the security level of instruction i. These notions are summarized in Figure 3.4.

The field of each class is assigned either a (fixed) security label $\langle \emptyset, l \rangle$ or the special symbol $\star$ for declaring the field polymorphic as explained below. The security label of fields is determined by a function $\mathsf{ft} : \mathbb{F} \to \kappa \cup \{\star\}$. Fields declared polymorphic adopt a level determined by the type system at the point where creation of its host object takes place. The field is given the security level of the context in which the corresponding `new` instruction is executed. Given that multiple instances of the same class may have possibly different security labels for the same fields, we need to perform some bookkeeping. This is achieved by associating with each location a *symbolic location*.

Heap types map symbolic locations to expressions of the form $[f_1 : \kappa_1, \ldots, f_n : \kappa_n]$ called *object types*. An object type gives information on the security label of each field. We write $T(a, f)$ for the label associated to field f of the object type $T(a)$.

Definition 3.8 (Method). *A method M is an expression of the form $((x_1 : \kappa_1, \ldots, x_n : \kappa_n, \kappa_r), B)$, abbreviated $((\overrightarrow{x : \kappa}, \kappa_r), B)$, where $\kappa_1, \ldots, \kappa_n$ are the security labels of the formal parameters $x_1, \ldots, x_n$, κ_r is the security label of the value returned by the method, and B is a program referred to as its body.*

Methods $M = ((\overrightarrow{x : \kappa}, \kappa_r), B)$ are typed by means of a *typing judgement*: $\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T} \rhd M$, where $(\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T})$ is a typing context for M. The method M shall be considered well-typed if such a judgement is derivable by means of appropriate type schemes.

Two further ingredients are required before formulating the type system: (1) a notion of subtyping for frame, heap and stack types; and (2) the availability of control dependence region information.

3.3.1 Subtyping Frame, Heap and Stack Types

The notion of subtyping for frame array, heap and stack types is presented in Figure 3.5.

Frame types are compared pointwise using the ordering on security labels. For example, if $\mathbb{X} = \{x, y\}$, $V_1 = \{x \mapsto \mathtt{L}, y \mapsto \mathtt{H}\}$ and $V_2 = \{x \mapsto \mathtt{H}, y \mapsto \mathtt{H}\}$, then $V_1 \leq V_2$.

A heap type T_1 is a subtype of T_2 if the domain of T_1 is included in that of T_2; for all field f of a symbolic location a, the security level of $T_1(a, f)$ and $T_2(a, f)$ are equal and the symbolic location sets of the security label of $T_1(a, f)$ is included in those of $T_2(a, f)$. For example, let

$$T_1 = \{(a, [f_1 : \langle \emptyset, \mathtt{H} \rangle, f_2 : \langle \{b\}, \mathtt{L} \rangle]), (b, [f_1 : \langle \emptyset, \mathtt{H} \rangle])\} \text{ and}$$
$$T_2 = \{(a, [f_1 : \langle \emptyset, \mathtt{H} \rangle, f_2 : \langle \{c, b\}, \mathtt{L} \rangle]), (b, [f_1 : \langle \emptyset, \mathtt{H} \rangle])(c, [f_1 : \langle \emptyset, \mathtt{H} \rangle])\}$$

We note that $\mathsf{Dom}(T_1) = \{a, b\}$ and $\mathsf{Dom}(T_2) = \{a, b, c\}$; $T_1(a, f_1) = \langle \emptyset, \mathtt{H} \rangle$ and $T_2(a, f_1) = \langle \emptyset, \mathtt{H} \rangle$; $T_1(a, f_2) = \langle \{b\}, \mathtt{L} \rangle$ and $T_2(a, f_2) = \langle \{c, b\}, \mathtt{L} \rangle$; and, $T_1(b, f_1) = \langle \emptyset, \mathtt{H} \rangle$ and $T_2(b, f_1) = \langle \emptyset, \mathtt{H} \rangle$ then $T_1 \leq T_2$.

Stack types may be only compared if they are of the same size. We use $\|S\|$ for the length of stack S. Furthermore, we allow depth subtyping at position i provided that $S(i)$ is at least l. $S_1 \leq_l S_2$ asserts that stack type S_1 is a subtype of S_2 at level l. This last requirement is due to the security level of low stack element can not vary in a high context. If $l = \mathtt{H}$, $S_1(2) = \mathtt{L}$ and $S_2(2) = \mathtt{H}$ then $S_1 \not\leq_l S_2$.

$$\text{(SubTyp-Frame)}\quad \frac{\forall x \in \mathbb{X}. V_1(x) \sqsubseteq V_2(x)}{V_1 \leq V_2}$$

$$\text{(SubTyp-Heap)}\quad \frac{\begin{array}{c}\mathsf{Dom}(T_1) \subseteq \mathsf{Dom}(T_2) \\ \forall a \in \mathsf{Dom}(T_1).\mathsf{Dom}(T_1(a)) = \mathsf{Dom}(T_2(a)) \wedge \forall f \in \mathsf{Dom}(T_1(a)).T_1(a,f) \stackrel{\sim}{\sqsubseteq} T_2(a,f)\end{array}}{T_1 \leq T_2}$$

$$\text{(SubTyp-Stack)}\quad \frac{\begin{array}{c}\|S_1\| = \|S_2\| = n \\ \forall j \in 0..n-1. \begin{cases} S_1(j) \sqsubseteq S_2(j) \text{ if } l \sqsubseteq S_1(j), \\ S_1(j) = S_2(j) \text{ otherwise}\end{cases}\end{array}}{S_1 \leq_l S_2}$$

Fig. 3.5. Subtyping frame, heap and stack types

3.3.2 Control Dependence Region

High-level languages have control-flow constructs that explicitly state dependency. For example, from

$$\texttt{while } x \texttt{ begin } c1;c2 \texttt{ end}$$

one easily deduces that both `c1` and `c2` depend on `x`. Given that such constructs are absent from JVM^s, as is the case in most low-level languages, and that they may be the source of unwanted information flows, our type system requires this information to be supplied (cf. Sec.3.3.5). Let $\mathsf{Dom}(B)^\sharp$ denote the set of program points of B where branching instructions occur (i.e. $\mathsf{Dom}(B)^\sharp = \{k \in \mathsf{Dom}(B) \mid B(k) = \texttt{if } i\}$). We assume given two functions ($\wp$ below denotes the powerset operator):

- region: $\mathsf{Dom}(B)^\sharp \to \wp(\mathsf{Dom}(B))$
- jun: $\mathsf{Dom}(B)^\sharp \to \mathsf{Dom}(B)$

The first computes the *control dependence region*, an overapproximation of the range of branching instructions and the second the unique *junction point* of a branching instruction at a given program point. In the Figure 3.6 shows an example of these functions.

Following previous work [21, 26, 24], we only require some abstract properties of these functions to hold. We need the notion of successor relation to formulate these properties.

Definition 3.9. *The successor relation* $\mapsto \subseteq \mathsf{Dom}(B) \times \mathsf{Dom}(B)$ *of a method* B *is defined by the clauses:*

- *If* $B(i) = \texttt{goto } i'$, *then* $i \mapsto i'$;
- *If* $B(i) = \texttt{if } i'$, *then* $i \mapsto i+1$ *and* $i \mapsto i'$;
- *If* $B(i) = \texttt{return}$, *then* $i \not\mapsto$ *(i has no successors);*
- *Otherwise,* $i \mapsto i+1$;

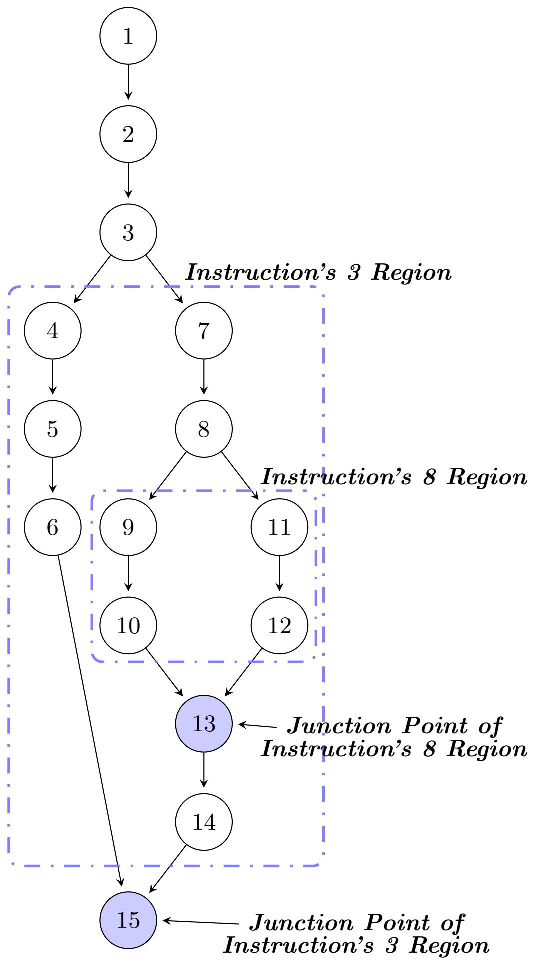
A small set of requirements, the *safe over approximation property* or *SOAP*, on region and jun suffice for the proof of noninterference. The SOAP state some basic properties on how the successor relation, region and jun are related.

Property 3.10 (SOAP). Let $i \in \mathsf{Dom}(B)^\sharp$.

1. If $i \mapsto i'$, then $i' \in \mathsf{region}(i)$.
2. If $i' \mapsto i''$ and $i' \in \mathsf{region}(i)$, then $i'' \in \mathsf{region}(i)$ or $i'' = \mathsf{jun}(i)$.
3. If $i \mapsto^* i'$ and $i' \in \mathsf{Dom}(B)^\sharp \cap \mathsf{region}(i)$, then $\mathsf{region}(i') \subseteq \mathsf{region}(i)$.

```
1   push   0
2   load   x
3   if   7
4   push   0
5   store   y
6   goto   15
7   load   z
8   if   11
9   push   1
10  goto   13
11  push   2
12  goto   13
13  store   y
14  goto   15
15  return
```

(a) Program


Instruction's 3 Region
Instruction's 8 Region
Junction Point of Instruction's 8 Region
Junction Point of Instruction's 3 Region
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

(b) Regions and Junction Point

Fig. 3.6. Example of Region and Junction Point

4. If $\mathsf{jun}(i)$ is defined, then $\mathsf{jun}(i) \notin \mathsf{region}(i)$ and $\forall i'$ such that $i \mapsto^* i'$, either $i' \mapsto^* \mathsf{jun}(i)$ or $\mathsf{jun}(i) \mapsto^* i'$.

Intuitively, successors i' of a branching instruction i should be in the same region. As for successors i'' of an instruction i' with $i' \in \mathsf{region}(i)$, either i'' should be in the region associated to i or be a junction point for this region. Regarding the third item, we write $\mapsto^*$ for the reflexive, transitive closure of $\mapsto$ and say that i' is an *indirect successor* of i when $i \mapsto^* i'$. This item reads as follows: if i' is the program point of a branching instruction that is an indirect successor of a branching instruction i that has not left the region of i, then the region of i' is wholly inside that of i. The final item states that the junction point of a region are not part of the region and, moreover, all indirect successors of the program point i of a branching instruction either come before or after (in terms of the indirect successor relation) the junction point of the region of i.

Remark 3.11. In what follows we shall adopt two assumptions that, without sacrificing expressiveness, simplify our analysis and soundness proof. First we assume a method body B has a unique `return` instruction. Second, that predecessors of a junction point are always `goto` instructions (i.e. $i \in \mathsf{region}(k)$ and $i \mapsto i'$ and $i' = \mathsf{jun}(k)$ implies $B(i) = \texttt{goto}\ i'$).

3.3.3 Typing Schemes

This section describes the typing schemes that define when a method is to be considered *well-typed*. We assume that this method is valid JVM code in the sense that it passes the bytecode verifier. For example, in an instruction such as `prim` $+$ we do not check whether the operands are indeed numbers. The typing schemes rely on previously supplied region and jun satisfying the SOAP properties, as discussed above.

Definition 3.12. *A method M is well-typed if there exists a typing context $(\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T})$ such that the type judgement*

$$\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T} \rhd M$$

is derivable using the following typing scheme

$$\frac{\begin{array}{c}\forall x_j \in \vec{x}.\mathcal{V}_1(x_j) = \kappa_j \qquad \mathcal{S}_1 = \epsilon \\ \forall x_j \in \vec{x}.\forall a \in \lceil \kappa_j \rceil . a \in \mathsf{Dom}(\mathcal{T}_1) \\ \forall i \in \mathsf{Dom}(B).\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T}, i \rhd ((\overrightarrow{x:\kappa},\kappa_r),B)\end{array}}{\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T} \rhd ((\overrightarrow{x:\kappa},\kappa_r),B)}$$

The variable array type $\mathcal{V}_1$ must provide labels for all parameters and must agree with the ones assigned to each of them by the declaration $\overrightarrow{x:\kappa}$. The stack is assumed to be empty on entry. Condition "$\forall x_j \in \vec{x}.\forall a \in \lceil \kappa_j \rceil . a \in \mathsf{Dom}(\mathcal{T}_1)$" ensures that, if the label of a parameter has a nonempty set of symbolic locations, then $\mathcal{T}_1$ does not leave out these references. Finally, all program points should be well-typed under the typing contexts $\mathcal{V},\mathcal{S},\mathcal{A}$ and $\mathcal{T}$. A program point i of B is well-typed if the judgement

$$\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T}, i \rhd ((\overrightarrow{x:\kappa},\kappa_r),B)$$

holds. This will be the case if it is derivable using the *typing schemes for instructions* of Figure 3.7. For ease of presentation, in the schemes of Figure 3.7 we have abbreviated $((\overrightarrow{x:\kappa},\kappa_r),B)$ with B. Note that there may be more than one typing context $(\mathcal{V},\mathcal{S},\mathcal{T},\mathcal{A})$ such that a program is well-typed. For reasons of technical convenience we assume, without loss of generality, that the chosen typing context is the least solution. A typing context C such that a program B is well-typed is the least solution if for all typing context $C_1,\cdots,C_n$ such that a program B is well-typed then $C = \prod_{i=1,\cdots,n} C_i$. Furthermore, we assume $\forall i \in \mathsf{Dom}(B).\mathcal{A}_1 \preceq \mathcal{A}_i$.

We now describe the typing schemes. They introduce a set of constraints between the frame types, stack types and heap types at different instructions in the program.

T-PrimOp requests that $\mathcal{S}_i$ have at least two elements on the top and that the top element of $\mathcal{S}_{i+1}$, the stack type of the successor instruction, have at least that of these elements. We write $\mathcal{S}_{i+1}(0)$ for the topmost element of the stack type $\mathcal{S}_{i+1}$ and $\mathcal{S}_{i+1}\backslash 0$ for $\mathcal{S}_{i+1}$ without the topmost element. Subtyping (rather than equality) for variable array, stack and heap types are required given that instruction $i+1$ may have other predecessors apart from i.

Concerning T-Str, since instruction `store` x does change local variables, $\mathcal{V}_i \setminus x$ must be a subtype of $\mathcal{V}_{i+1} \setminus x$ as stated by the third line. Furthermore, the value of local variable x is modified by stack top, then $\mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{V}_{i+1}(x) \cdot \mathcal{S}_{i+1}$ must hold. The fourth line states that the security level of the x value in follow instruction must be greater than or equal to the security level of current address.

In T-Load, the instruction `load` x does not change local variables, $\mathcal{V}_i$ must be a subtype of $\mathcal{V}_{i+1}$. Since the value of local variable x is pushed into the stack the security level of the pushed value must be greater than or equal to the security level of local variable x and to the security level of current address.

T-Ret simply requires that the label of the topmost element of the stack does not leak any information. Here, κ_r is the label of the result of the method body. Note that $\mathcal{A}_i$ is not resorted to given that by assumption there is a unique `return` instruction and this instruction is executed with $\mathcal{A}_i$ at low. Typing schemes T-Pop, T-Push and T-Goto follow similar lines.

Regarding T-If, we assume that `if` i' instructions have always two branches (in this case i' and $i+1$). Since control jumps to i' or $i+1$, $\mathcal{V}_i$ must be a subtype of $\mathcal{V}_{i'}$ and $\mathcal{V}_{i+1}$ as stated by the second line. $\mathcal{S}_i$

(T-PrimOp)
$$\frac{\begin{array}{l} B(i) = \texttt{prim}\ op \\ \kappa' \sqcup \kappa'' \sqsubseteq \mathcal{S}_{i+1}(0) \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \kappa' \cdot \kappa'' \cdot (\mathcal{S}_{i+1} \backslash 0) \\ \mathcal{A}_i \sqsubseteq \kappa', \kappa'' \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B}$$

(T-Str)
$$\frac{\begin{array}{l} B(i) = \texttt{store}\ x \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{V}_{i+1}(x) \cdot \mathcal{S}_{i+1} \\ \mathcal{V}_i \backslash x \leq \mathcal{V}_{i+1} \backslash x \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \\ \mathcal{A}_i \sqsubseteq \mathcal{V}_{i+1}(x) \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B}$$

(T-Load)
$$\frac{\begin{array}{l} B(i) = \texttt{load}\ x \\ \mathcal{A}_i \sqcup \mathcal{V}_i(x) \cdot \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B}$$

(T-Ret)
$$\frac{\begin{array}{l} B(i) = \texttt{return} \\ \mathcal{S}_i(0) \sqcup \mathcal{A}_i \sqsubseteq \kappa_r \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B}$$

(T-Pop)
$$\frac{\begin{array}{l} B(i) = \texttt{pop} \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \kappa \cdot \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \\ \mathcal{A}_i \sqsubseteq \kappa \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B}$$

(T-Push)
$$\frac{\begin{array}{l} B(i) = \texttt{push}\ n \\ \mathcal{A}_i \cdot \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B}$$

(T-Goto)
$$\frac{\begin{array}{l} B(i) = \texttt{goto}\ i' \\ \mathcal{V}_i \leq \mathcal{V}_{i'} \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\ \mathcal{T}_i \leq \mathcal{T}_{i'} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B}$$

(T-If)
$$\frac{\begin{array}{l} B(i) = \texttt{if}\ i' \\ \mathcal{V}_i \leq \mathcal{V}_{i+1}, \mathcal{V}_{i'} \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \kappa \cdot \mathcal{S}_{i+1}, \kappa \cdot \mathcal{S}_{i'} \\ \mathcal{A}_i \sqsubseteq \kappa \\ \mathcal{T}_i \leq \mathcal{T}_{i+1}, \mathcal{T}_{i'} \\ \forall k \in \texttt{region}(i). \kappa \sqsubseteq \mathcal{A}_k \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B}$$

(T-New)
$$\frac{\begin{array}{l} B(i) = \texttt{new}\ C \\ a = \mathsf{Fresh}(\mathcal{T}_i) \\ \mathcal{T}_i \oplus \{a \mapsto [f_1 : \texttt{ft}_{\mathcal{A}_i}(f_1), \dots, f_n : \texttt{ft}_{\mathcal{A}_i}(f_n)]\} \leq \mathcal{T}_{i+1} \\ \langle \{a\}, \mathcal{A}_i \rangle \cdot \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B}$$

(T-PtFld)
$$\frac{\begin{array}{l} B(i) = \texttt{putfield}\ f \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \langle R_1, l_1 \rangle \cdot \langle R, l \rangle \cdot \mathcal{S}_{i+1} \\ l_1 \sqcup l \sqsubseteq \bigsqcap_{a \in R} \mathcal{T}_i(a, f) \\ \textit{for all } a \in R. \mathcal{T}_{i+1}(a, f) = \langle R_3, l_2 \rangle, \\ \qquad\qquad \textit{where } \mathcal{T}_i(a, f) = \langle \_, l_2 \rangle \wedge R_1 \subseteq R_3 \\ \mathcal{A}_i \sqsubseteq l, l_1 \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \backslash \{(a, f) \mid a \in R\} \leq \mathcal{T}_{i+1} \backslash \{(a, f) \mid a \in R\} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B}$$

(T-GtFld)
$$\frac{\begin{array}{l} B(i) = \texttt{getfield}\ f \\ \mathcal{S}_i(0) = \langle R, l \rangle \\ \kappa = \bigsqcup_{a \in R} \mathcal{T}_i(a, f) \\ l \sqcup \kappa \sqcup \mathcal{A}_i \cdot (\mathcal{S}_i \backslash 0) \leq_{\mathcal{A}_i} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B}$$

Fig. 3.7. Typing Schemes

must be a subtype of $\kappa \cdot \mathcal{S}_{i'}$ and $\kappa \cdot \mathcal{S}_{i+1}$. Moreover, the security level of $\mathsf{region}(i)$ must be greater than or equal to the security level κ of the value inspected at i.

In order to type `new` C a fresh symbolic location is created. The $\mathsf{Fresh}(\mathcal{T}_i)$ function return a symbolic location not used in $\mathcal{T}_i$. The heap type of the successor of i is required to define an object type for it. This object type assigns a security label to each field according to $\texttt{ft}_{\mathcal{A}_i}$. For those fields declared polymorphic, the level $\mathcal{A}_i$ is assigned.

T-PtFld requires that no information about the value written to field f (l_1), the reference of the object (l) nor the current security context ($\mathcal{A}_i$) be leaked. The security level of the field f does not change (the level is fixed throughout the execution). However, the symbolic locations associated with the attribute (f) are updated with the value symbol locations (R_1). Note that label $\langle R, l \rangle$ may contain multiple symbolic references in R. For example, this would be the case if a new object was created in two different branches of a conditional before merging and then executing the `putfield` instruction.

The scheme T-GtFld requires that information about the current security context ($\mathcal{A}_i$), the reference of the object (l) and the value read to the field (κ) is pushed on top of the stack.

3.3.4 Type-Checking Examples

The combination of T-If scheme with T-Store, T-Load, T-PtFld or T-GtFld allows to prevent indirect flows, as we now exemplify:

Example 3.13.

```
1  load  $x_L$
2  load  $y_H$
3  if  7
4  push  0
5  putfield  $f_L$
6  goto  10
7  push  1
8  putfield  $f_L$
9  goto  10
10 return
```

By requiring the updated field level to be greater or equal than the context level in the T-PtFld rule and by requiring a global constraint on the security context in the T-If rule, the type system rejects this program. Since instructions at 5 and 8 are in region(3) (hence are under the influence of high `if`), these program points must be H ($\mathcal{A}_5 = \mathcal{A}_8 = \mathtt{H}$).

The condition: $S_i \leq_{\mathcal{A}_i} S_{i+1}$ on rules T-PrimOp, T-If,T-Pop,T-Str and T-PtFld determines the operand stack elements that will be popped in a high context. This condition avoids implicit flows caused by the manipulation of the operand stack. In high contexts one cannot remove low stack elements.

Example 3.14. In the following program, the final value of the top of the stack in `return` (3 or 4) depends on the initial value of $y_{\mathtt{H}}$. The problem is caused by an arithmetic instruction that manipulates (removes a low value) the operand stack in the scope of a high `if` instruction.

```
1 push  3
2 load  $y_H$
3 if  7
4 push  1
5 prim  +
6 goto  8
7 goto  8
8 return
```

Subtyping on heap types is required to accomodate new symbolic locations.

Example 3.15. Consider the program fragment:

```
1 new  C
2 if  i
3 new  D
4 goto  2
  ...
```

$\mathcal{T}_2$ will assign symbolic locations to both the new C object *and* the new D object. The reason for the latter is the `goto` instruction which folds back the heap type assignment to $\mathcal{T}_2$.

Example 3.16 shows the result of type checking.

Example 3.16. Consider the following method:

```
1  push  0
2  load  x
3  if  7
4  push  0
5  store  y
6  goto  10
7  push  1
8  store  y
9  goto  10
10 return
```

The final value of y (0 or 1) depends of the value of x (push on stack in the instruction 2) evaluated in the instruction 3. The type checking algorithm generates the following constraints, with $\kappa_r = \mathtt{L}$ and $\mathsf{region}(3) = \{4, 5, 6, 7, 8, 9\}$:

i	$B(i)$	$\mathcal{V}_i(x)$	$\mathcal{V}_i(y)$	$\mathcal{S}_i$	$\mathcal{A}_i$	*Constraints Generated*
1	push 0	H	H	ϵ	L	$\mathtt{L} \cdot \epsilon \leq_{\mathtt{L}} \beta_1 \cdot \epsilon$
2	load x	H	H	$\beta_1 \cdot \epsilon$	δ_2	$\mathtt{H} \cdot \beta_1 \cdot \epsilon \leq_{\delta_2} \beta_2 \cdot \beta_1 \cdot \epsilon \quad \delta_2 \sqsubseteq \beta_2$
3	if 7	H	H	$\beta_2 \cdot \beta_1 \cdot \epsilon$	δ_3	$\forall i \in \{4,5,6,7,8,9\}. \beta_2 \sqsubseteq \mathcal{A}_i$
4	push 0	H	H	$\beta_1 \cdot \epsilon$	δ_4	$\delta_4 \cdot \beta_1 \cdot \epsilon \leq_{\delta_4} \beta_3 \cdot \beta_1 \cdot \epsilon$
5	store y	H	H	$\beta_3 \cdot \beta_1 \cdot \epsilon$	δ_5	$\beta_3 \cdot \beta_1 \cdot \epsilon \leq_{\delta_5} \gamma_1 \cdot \beta_1 \cdot \epsilon \quad \delta_5 \sqsubseteq \gamma_1$
6	goto 10	H	γ_1	$\beta_1 \cdot \epsilon$	δ_6	$\mathcal{V}_6(y) \sqsubseteq \mathcal{V}_{10}(y)$
7	push 1	H	γ_1	$\beta_1 \cdot \epsilon$	δ_7	$\delta_7 \cdot \beta_1 \cdot \epsilon \leq_{\delta_7} \beta_4 \cdot \beta_1 \cdot \epsilon$
8	store y	H	γ_1	$\beta_4 \cdot \beta_1 \cdot \epsilon$	δ_8	$\beta_4 \cdot \beta_1 \cdot \epsilon \leq_{\delta_8} \gamma_2 \cdot \beta_1 \cdot \epsilon \quad \delta_8 \sqsubseteq \gamma_2$
9	goto 10	H	γ_2	$\beta_1 \cdot \epsilon$	δ_9	$\mathcal{V}_6(y) \sqsubseteq \mathcal{V}_{10}(y)$
10	store z	H	γ_2	$\beta_1 \cdot \epsilon$	δ_{10}	$\beta_1 \sqcup \delta_{10} \sqsubseteq \kappa_r$

The constraints are satisfiable by:

$$\beta_1, \delta_2, \delta_3, \delta_{10} = \mathtt{L} \qquad \beta_2, \beta_3, \beta_4, \delta_4, \delta_5, \delta_6, \delta_7, \delta_8, \delta_9, \gamma_1, \gamma_2 = \mathtt{H}$$

3.3.5 Implementing Type Checking

The type checking algorithm works in two phases:

1. first it collects constraints dictated by the typing schemes of each instruction;
2. second it solves the set of inequality constraints obtained in the first step. The least solution of this set of constraints is computed. Next, it verifies that all constraints are satisfied with the solution obtained.

Typing schemes were already covered in previous sections. Now, we explain how we compute the least solution for the constraints.

The Solution of Set Constraints

We call terms to expressions of the form:

$$\begin{array}{lll} var & ::= \beta_i \mid \delta_i \mid \gamma_i \mid c_i & \text{Variables} \\ t & ::= \mathtt{L} \mid \mathtt{H} \mid \langle R, t \rangle \mid var \mid t \sqcup t \mid t \sqcap t & \text{Terms} \end{array}$$

The least solution of a set of constraints is computed by the algorithm in Figure 3.8. This algorithm is a version of the classic *Abstract Worklist Algorithm* [90] for solving sets of inequalities. The algorithm uses:

- A set of constraints $t_1 \sqsubseteq c_1, \cdots, t_N \sqsubseteq c_N$, named `C`, of input.
- A worklist `W`.
- An auxiliary array of sets of constraints, `infl`, which records the constraints that depend on each variable.
- `Analysis` is the array that represents the solution. A solution is a total function that assigns to each variable a security label.
- The function $\texttt{eval}(t, \texttt{Analysis})$ evaluates term t according to the array `Analysis`.

The algorithm initializes the worklist `W` with the set of constraints. The initial solution for each element in `Analysis` is $\bot = \langle \emptyset, \texttt{L} \rangle$, the least element. And, after the initialisation $\texttt{infl}[c] = \{t' \sqsubseteq c' \in \texttt{C} \mid c \textit{ appears in } t'\}$.

The worklist is then processed until it is empty. In each iteration one constraint is computed and all constraints that depend on it are added to the list for recomputing if the level differs (is greater) from the one computed earlier.

```
INPUT:          A system C of constraints: $t_1 \sqsubseteq c_1, \cdots, t_N \sqsubseteq c_N$

OUTPUT:         The least solution: Analysis

INITIALISATION: (of W, Analysis and infl)
  W := $\emptyset$
  for all $t \sqsubseteq c \in$ C do
    W := W $\cup \{t \sqsubseteq c\}$
    Analysis[$c$] := $\bot$
    infl[$c$] := $\emptyset$
  end for
  for all $t \sqsubseteq c \in$ C do
    for all $c'$ in $t$ do
      infl[$c'$] := infl[$c'$] $\cup \{t \sqsubseteq c\}$
    end for
  end for

ITERATION:      (updating W and Analysis)
  while W $\neq \emptyset$ do
    $t \sqsubseteq c$ := choose some $t \sqsubseteq c \in$ W
    W := W $\setminus \{t \sqsubseteq c\}$
    new := eval($t$, Analysis)
    if new $\not\sqsubseteq$ Analysis[$c$] then
      Analysis[$c$] := Analysis[$c$] $\sqcup$ new
      for all $t' \sqsubseteq c' \in$ infl[$c$] do
        W := W $\cup \{t' \sqsubseteq c'\}$
      end for
    end if
  end while
```

Fig. 3.8. The Constraints Solver Algorithm

We need the set of constraints to be as follows: $t_1 \sqsubseteq c_1, \cdots, t_N \sqsubseteq c_N$, where $c_1, \cdots, c_N$ are variables and $t_1, \cdots, t_N$ are terms. Then we rewrite the constraints generated by the type checker with the algorithm of Figure 3.9.

```
INPUT: The set of constraints $\mathcal{C}$ generated by the type schemes.

OUTPUT: A system $\mathtt{C}$ of constraints: $t_1 \sqsubseteq c_1, \cdots, t_N \sqsubseteq c_N$

  $\mathtt{C} := \emptyset$

  for all $\mathcal{S} \leq_l \mathcal{S}' \in \mathcal{C}$ do
    for all $j \in \{0 \cdots \|\mathcal{S}\|\}$ do
      $\mathtt{C} := \mathtt{C} \cup \{\mathcal{S}(j) \sqsubseteq \mathcal{S}'(j)\}$
    end for
  end for

  for all $\mathcal{V} \leq \mathcal{V}' \in \mathcal{C}$ do
    for all $x \in \mathbb{X}$ do
      $\mathtt{C} := \mathtt{C} \cup \{\mathcal{V}(x) \sqsubseteq \mathcal{V}'(x)\}$
    end for
  end for

  for all $\mathcal{T} \leq \mathcal{T}' \in \mathcal{C}$ do
    for all $a \in \mathsf{Dom}(\mathcal{T})$ do
      for all $f \in \mathsf{Dom}(\mathcal{T}(a))$ do
        $\mathtt{C} := \mathtt{C} \cup \{\mathcal{T}(a, f) \underset{\approx}{\sqsubseteq} \mathcal{T}'(a, f)\}$
      end for
    end for
  end for

  for all $(t \sqsubseteq t'$ or $t = t') \in \mathcal{C}$ do
    $\mathtt{C} := \mathtt{C} \cup \{t \sqsubseteq t'\}$
  end for

  for all $t \sqsubseteq \kappa \sqcap t' \in \mathtt{C}$ do
    $\mathtt{C} := (\mathtt{C} \setminus \{t \sqsubseteq \kappa \sqcap t'\}) \cup \{t \sqsubseteq \kappa\} \cup \{t \sqsubseteq t'\}$
  end for
```

Fig. 3.9. The Rewrite Constraints Algorithm

Let us return to the Example 3.16. We have the constraints set generated by the type checker:

$$\begin{gathered}
\mathtt{L} \cdot \epsilon \leq_{\mathtt{L}} \beta_1 \cdot \epsilon \\
\mathtt{H} \cdot \beta_1 \cdot \epsilon \leq_{\gamma_2} \beta_2 \cdot \beta_1 \cdot \epsilon \quad \delta_2 \sqsubseteq \beta_2 \\
\forall i \in \{4, 5, 6, 7, 8, 9\}.\beta_2 \sqsubseteq \mathcal{A}_i \\
\delta_4 \cdot \beta_1 \cdot \epsilon \leq_{\gamma_4} \beta_3 \cdot \beta_1 \cdot \epsilon \\
\beta_3 \cdot \beta_1 \cdot \epsilon \leq_{\gamma_5} \gamma_1 \cdot \beta_1 \cdot \epsilon \quad \delta_5 \sqsubseteq \gamma_1 \\
\mathcal{V}_6(y) \sqsubseteq \mathcal{V}_{10}(y) \\
\delta_7 \cdot \beta_1 \cdot \epsilon \leq_{\gamma_7} \beta_4 \cdot \beta_1 \cdot \epsilon \\
\beta_4 \cdot \beta_1 \cdot \epsilon \leq_{\gamma_8} \gamma_2 \cdot \beta_1 \cdot \epsilon \quad \delta_8 \sqsubseteq \gamma_2 \\
\mathcal{V}_6(y) \sqsubseteq \mathcal{V}_{10}(y) \\
\beta_1 \sqcup \delta_{10} \sqsubseteq \kappa_r
\end{gathered}$$

The Algorithm of Figure 3.9 rewrite the set of constraints, then we have:

$$\begin{gathered}
\mathtt{L} \sqsubseteq \beta_1 \\
\mathtt{H} \sqsubseteq \beta_2 \qquad \delta_2 \sqsubseteq \beta_2 \\
\beta_2 \sqsubseteq \delta_4, \delta_5, \delta_6, \delta_7, \delta_8, \delta_9 \\
\delta_4 \sqsubseteq \beta_3 \\
\beta_3 \sqsubseteq \gamma_1 \qquad \delta_5 \sqsubseteq \gamma_1 \\
\gamma_1 \sqsubseteq \gamma_4 \\
\delta_7 \sqsubseteq \beta_4 \\
\beta_4 \sqsubseteq \gamma_2 \qquad \delta_8 \sqsubseteq \gamma_2 \\
\gamma_2 \sqsubseteq \gamma_4 \\
\beta_1 \sqcup \delta_{10} \sqsubseteq \mathtt{L}
\end{gathered}$$

We show how the algorithm obtains the solution of the constraints for a subset of the restrictions. We consider the following set of constraints $\mathtt{C} = \{\mathtt{L} \sqsubseteq \beta_1, \mathtt{H} \sqsubseteq \beta_2, \delta_2 \sqsubseteq \beta_2, \beta_2 \sqsubseteq \delta_4\}$. The Constraint Solver initialisation (Figure 3.8) produces:

$$\mathtt{W} = \mathtt{L} \sqsubseteq \beta_1, \mathtt{H} \sqsubseteq \beta_2, \delta_2 \sqsubseteq \beta_2, \beta_2 \sqsubseteq \delta_4$$

	β_1	β_2	δ_2	δ_4
`Analysis`	L	L	L	L
`infl`	$\emptyset$	$\beta_2 \sqsubseteq \delta_4$	$\delta_2 \sqsubseteq \beta_2$	$\emptyset$

The first iteration take a constraint, by example $\mathtt{L} \sqsubseteq \beta_1$. Then the first iteration result is

$$\mathtt{W} = \mathtt{H} \sqsubseteq \beta_2, \delta_2 \sqsubseteq \beta_2, \beta_2 \sqsubseteq \delta_4$$

	β_1	β_2	δ_2	δ_4
`Analysis`	L	L	L	L
`infl`	$\emptyset$	$\beta_2 \sqsubseteq \delta_4$	$\delta_2 \sqsubseteq \beta_2$	$\emptyset$

We note that since $\mathtt{eval}(\mathtt{L}, \mathtt{Analysis}) = \mathtt{L}$, then $\mathtt{Analysis}(\beta_1)$ does not change.
The following is $\mathtt{H} \sqsubseteq \beta_2$. We have $\mathtt{eval}(\mathtt{H}, \mathtt{Analysis}) = \mathtt{H}$, then $\mathtt{Analysis}(\beta_2)$ changes.

$$\mathtt{W} = \delta_2 \sqsubseteq \beta_2, \beta_2 \sqsubseteq \delta_4$$

	β_1	β_2	δ_2	δ_4
`Analysis`	L	H	L	L
`infl`	$\emptyset$	$\beta_2 \sqsubseteq \delta_4$	$\delta_2 \sqsubseteq \beta_2$	$\emptyset$

The third iteration does not generate changes. Now we take the last constraint, $\beta_2 \sqsubseteq \delta_4$. $\mathtt{eval}(\beta_2, \mathtt{Analysis}) = \mathtt{H}$ then $\mathtt{Analysis}(\delta_4)$ change. Then, the least solution is (the constraints are satisfiable by):

	β_1	β_2	δ_2	δ_4
`Analysis`	L	H	L	H

Some Considerations

Nielson et al. [90] prove that the algorithm computes the least solution. The assumptions required for the proof are:

1. $(\mathit{SecLabels}, \sqsubseteq)$ is a *complete lattice* that satisfies the *Ascending Chain Condition*. A *complete lattice* is a partially ordered set in which all subsets have both a supremum and an infimum. A complete lattice is said to satisfy the *Ascending Chain Condition* if every ascending chain of elements eventually terminates, or eventually stabilizes (that is, there is no infinite ascending chain).
2. The set of elements that we want to find the solution is finite.

These requirements are met so we can say that the algorithm computes the least solution. Furthermore, Nielson et al. [90] prove that the complexity of the algorithm for computing the least solution is $\mathbf{O}(h \cdot M^2 \cdot N)$, where N is the number of constraints, M is the largest size of the right hand sides of constraints and h is the height of $(SecLabels, \sqsubseteq)$.

Let n be the number of instructions, v the number of local variables of a method and let f be the number of fields of the symbolic locations into heap type. Furthermore, let b the maximun arity for a branching point. Then the size of the constraints generated from each instruction is $\mathbf{O}(b \cdot n + v + f)$, and hence the size of all the constraints generated is $\mathbf{O}(b \cdot n^2 + n \cdot v + n \cdot f)$. The constraints can be solved in quadratic time by using the linear-time algorithm of J. Rehof and T. Mogensen [93]. Therefore, the time complexity of type checking is quadratic [68].

Currently we assume that the region and jun functions verifying SOAP are supplied by the user. This is by no means necessary given that such information may be inferred by an analyzer using standard algorithms [15, 92, 33]. The complexity of computing regions is the same as the complexity of verifying the SOAP property, i.e. it can be done in quadratic time [24].

3.4 JVM^s_E - Adding Exceptions to JVM^s

This section extends our analysis to deal with *exceptions*. We briefly explain the JVM exception handling mechanism and then extend the operational semantics and inference schemes for JVM^s accordingly.

3.4.1 JVM Exception Handling

In Java, throwing an exception results in an immediate non-local transfer of control from the point where the exception has been thrown. This transfer of control may abruptly complete, one by one, multiple statements, constructor invocations, static and field initializer evaluations, and method invocations. The process continues until a `catch` clause is found that handles the thrown value. If no such clause can be found, the current program exits. As implemented by the JVM, each `try-catch` clause of a method is represented by an exception handler.An exception handler specifies the range of instructions into the JVM code implementing the method for which the exception handler is active, describes the type of exception that the exception handler is able to handle, and specifies the location of the code that is to handle that exception. An exception matches an exception handler if the pc of the instruction that caused the exception is in the range of pcs of the exception handler and the exception type is the same class of the class of exception that the exception handler handles. In the Example 3.17 we show a program with their exception handler.

Example 3.17.

```
try{
    y_H.f = v;
    y_H.f_1 = v;
}catch(E e){
    this_L.f_2 = v;
};
return v;
```

(a) Program

```
 1 load y_H
 2 push v
 3 putfield f
 4 load y_H
 5 push v
 6 putfield f_1
 7 push v
 8 return
 9 load this_L
10 push v
11 putfield f_2
12 goto 7
```

(b) Bytecode

The exception table consists of a single handler $(1,6,9)$*, that says that all program points in the interval* $[1,6]$ *are handled by the code starting at program point* 9*.*

When an exception is thrown, the JVM searches for a matching exception handler in the current method. If a matching exception handler is found, the system branches to the exception handling code specified by the matched handler. If no such exception handler is found in the current method, the current method invocation completes abruptly. On abrupt completion, the operand stack and local variables of the current method invocation are discarded, and its frame is popped, reinstating the frame of the invoking method. The exception is then re-thrown in the context of the invoker's frame and so on, continuing up the method invocation chain. If no suitable exception handler is found before the top of the method invocation chain is reached, the execution of the program in which the exception was thrown is terminated [75].

3.4.2 JVM^s_E

In the presence of exceptions, information leakage can be caused by new forms of implicit flows, as we now illustrate.

Example 3.18. Consider the following program that assigns v_1 to field f_1 of the object stored in y and v_2 to field f_2 of the object stored in x.

```
1 load      $y_{\mathtt{H}}$
2 push      $v_1$
3 putfield  ${f_1}_{\mathtt{H}}$
4 load      $x_{\mathtt{L}}$
5 push      $v_2$
6 putfield  ${f_2}_{\mathtt{L}}$
7 push      $v$
8 return
```

This program is interferent if y and f_1 are $\mathtt{H}$, x and f_2 is $\mathtt{L}$ and the exception table is $(1,7,8)$ (meaning the handler for a **Throwable** exception commencing at instruction 8 applies to instruction range 1 to 7). In a normal execution field f_2 is assigned the value v_2, while in an exceptional case (y contains a null reference) instructions 4, 5 and 6 are never reached. Since f_2 is low, it is observable and hence there is an implicit flow from $y_{\mathtt{H}}$ to f_2.

In the case that the handler is not defined and κ_r is $\mathtt{L}$, then this program is interferent since the exception that is raised is high (the object y is high).

Example 3.17 is another case of an interferent program. The assignment to $y_{\mathtt{H}}.f = v$ may throw an exception, skipping the assignment to $y_{\mathtt{H}}.f_1 = v$, and causing an assignment to $this_{\mathtt{L}}.f_2 = v$.

Operational semantics of JVM^s_E. The operational semantics of JVM^s_E is obtained by adding the reduction rules of Fig. 3.10 to those of JVM^s.

In order to simplify our presentation we assume the two following requirements. The first is a unique class **Throwable** in $\mathbb{C}$ of throwable exceptions. An instance of this class is created upon attempting to write to (or read from) a field whose object reference is *null*. Also, it should be the class of the topmost object of the operand stack just before a `throw` instruction (although we do not check this given the assumption that the method passes the JVM bytecode verifier). The other requirement is that all methods return a value. That is, if a method invocation terminates abruptly, then the value returned is a reference to a throwable object.

We assume given a (partial) function $\mathsf{Handler}$ that indicates whether a handler is available at a given program point. For example, if a `putfield` instruction appears at i, then $\mathsf{Handler}(i) = i'$ indicates that the handler for a *null* pointer reference is located at i'. If i' is not relevant, then we write $\mathsf{Handler}(i)\downarrow$. We write $\mathsf{Handler}(i)\uparrow$ if $\mathsf{Handler}(i)$ is undefined.

For each program point i such that $\mathsf{Handler}(i)\uparrow$ we assume the existence of a dummy program point err_i to which control is transferred in the case that i generates an exception. To model exceptional behavior we use states of the form $\langle err_i, \alpha, \langle o\rangle \cdot \sigma, \eta\rangle$, where o is a reference to a throwable object. This allows us to model the attacker having access to the machine state on abnormal (i.e. due to an unhandled exception) termination. Additionally, it simplifies the notion of noninterference (indeed, our previous formulation applies to JVM^s_E too). Note that the use of multiple dummy program points also provides support for further precision in our analysis given that the memory state at different program points may differ substantially.

Upon raising an exception and assuming a handler is available at a given program point, the contents of the operand stack is assumed to be cleared. Note, however, that the information in the heap is maintained.

Furthermore, we assume the existence of a final program point p_f. The control is transferred from all dummy program points and the `return` instruction to final program point p_f. This allows us to model the attacker having access to the final machine state. We note that the final program point p_f may be a junction point. In that case it is the only one we do not assume that is preceded by `goto` instructions. Indeed, the final program point p_f is either preceded by the `return` instruction and/or dummy program points. Now, *final states* are referred to as expressions of the form $\langle p_f, v, \eta\rangle$.

$$\text{(R-PtFldH)}\quad \frac{\begin{array}{c} B(i) = \mathtt{putfield}\ f \quad o = null \\ o' = \mathsf{Fresh}(\eta) \quad \mathsf{Handler}(i) = i' \end{array}}{\langle i, \alpha, v \cdot o \cdot \sigma, \eta\rangle \longrightarrow_B \langle i', \alpha, o' \cdot \epsilon, \eta \oplus \{o' \mapsto \mathsf{Default}(\mathbf{Throwable})\}\rangle}$$

$$\text{(R-PtFldNoH)}\quad \frac{\begin{array}{c} B(i) = \mathtt{putfield}\ f \quad o = null \\ o' = \mathsf{Fresh}(\eta) \quad \mathsf{Handler}(i)\uparrow \end{array}}{\langle i, \alpha, v \cdot o \cdot \sigma, \eta\rangle \longrightarrow_B \langle err_i, \alpha, \langle o'\rangle \cdot \sigma, \eta \oplus \{o' \mapsto \mathsf{Default}(\mathbf{Throwable})\}\rangle}$$

$$\text{(R-GtFldH)}\quad \frac{\begin{array}{c} B(i) = \mathtt{getfield}\ f \quad o = null \\ o' = \mathsf{Fresh}(\eta) \quad \mathsf{Handler}(i) = i' \end{array}}{\langle i, \alpha, o \cdot \sigma, \eta\rangle \longrightarrow_B \langle i', \alpha, o' \cdot \epsilon, \eta \oplus \{o' \mapsto \mathsf{Default}(\mathbf{Throwable})\}\rangle}$$

$$\text{(R-GtFldNoH)}\quad \frac{\begin{array}{c} B(i) = \mathtt{getfield}\ f \quad o = null \\ o' = \mathsf{Fresh}(\eta) \quad \mathsf{Handler}(i)\uparrow \end{array}}{\langle i, \alpha, o \cdot \sigma, \eta\rangle \longrightarrow_B \langle err_i, \alpha, \langle o'\rangle \cdot \sigma, \eta \oplus \{o' \mapsto \mathsf{Default}(\mathbf{Throwable})\}\rangle}$$

$$\text{(R-ThrH)}\quad \frac{B(i) = \mathtt{throw} \quad \mathsf{Handler}(i) = i'}{\langle i, \alpha, o \cdot \sigma, \eta\rangle \longrightarrow_B \langle i', \alpha, o \cdot \epsilon, \eta\rangle} \qquad \text{(R-ThrNoH)}\quad \frac{B(i) = \mathtt{throw} \quad \mathsf{Handler}(i)\uparrow}{\langle i, \alpha, o \cdot \sigma, \eta\rangle \longrightarrow_B \langle err_i, \alpha, \langle o\rangle \cdot \sigma, \eta\rangle}$$

$$\text{(R-Erri)}\quad \frac{}{\langle err_i, \alpha, \langle o\rangle \cdot \sigma, \eta\rangle \longrightarrow_B \langle p_f, \langle o\rangle, \eta\rangle} \qquad \text{(R-Ret)}\quad \frac{B(i) = \mathtt{return}}{\langle i, \alpha, v \cdot \sigma, \eta\rangle \longrightarrow_B \langle p_f, v, \eta\rangle}$$

Fig. 3.10. Additional Reduction Schemes for Exception Handling

3.4.3 Typing Schemes

Given that exceptions are a form of conditional jumps control dependence regions must be extended. The extension is rather straightforward. First the set of conditional control points is modified as follows:

$$\mathsf{Dom}(B)^{\sharp} = \{k \in \mathsf{Dom}(B) \mid B(k) = \texttt{if}\ i \text{ or } B(k) = \texttt{putfield}\ f \text{ or } B(k) = \texttt{getfield}\ f\}$$

.

Note that `throw` is not a conditional jump and hence not included.

Then, we extend the successor relation (Def. 3.9) to include the possible change of control flow:

- If $B(i) = \texttt{throw}$ and $\mathsf{Handler}(i) = i'$, then $i \mapsto i'$, otherwise $i \mapsto err_i$;
- If $B(i) = \texttt{putfield}\ f$, then $i \mapsto i+1$ and if $\mathsf{Handler}(i) = i'$, then $i \mapsto i'$, otherwise $i \mapsto err_i$;
- If $B(i) = \texttt{getfield}\ f$, then $i \mapsto i+1$ and if $\mathsf{Handler}(i) = i'$, then $i \mapsto i'$, otherwise $i \mapsto err_i$;
- If $B(i) = \texttt{return}$, then $i \mapsto p_f$;
- If $B(err_i)$, then $err_i \mapsto p_f$.

The SOAP properties, however, are not modified since that notion is already parametrized by $\mathsf{Dom}(B)^{\sharp}$. But, we need extend SOAP to p_f:

- $\forall i \in \mathsf{Dom}(B).p_f \notin \mathsf{region}(i)$.

The additional typing schemes are presented in Figure 3.11. The main requirement in scheme T-ThrNoH, the typing scheme for the instruction `throw` when no handler is available, is $\mathcal{A}(i) \sqcup \mathcal{S}_i(0) \sqsubseteq \kappa_r$. In the case of an unhandled exception, the occurrence of the exception should not leak the level of the context. For example, if the level of the return value (κ_r) is `L`, then $\mathcal{A}_i$ should be low too for otherwise the occurrence or not of the exception could reveal the secret value on which an instruction branched, whose region includes the `throw` instruction. Likewise, the instance of the exception that is raised may be seen as the returned error value. In the case that there is a handler, then T-ThrH deals with the `throw` instruction as a unconditional jump to the handler. Furthermore, the operand stack are discarded and the object security level is pushed on the stack.

The `putfield` instruction requires three typing schemes. T-PtFld and T-PtFldH types a `putfield` at a program point that has a handler. T-PtFld and T-PtFldNoH are applied if the program point has a no handler. In the case of a `putfield` instruction located at a program point for which there is a handler, when control is transferred to this handler the operand stack is assumed to be purged and a new instance of **Throwable** created. This object's reference must be at least the level of the null reference and the context in which the `putfield` is executed. Most notably, the level of all items on the stack must be assumed high since its contents is available for inspection after an exception (i.e. the fact that it has been emptied is observable). It should be mentioned that this does not force all items of the stack of ancestor instructions to be high thanks to subtyping of stack types. The typing scheme T-PtFldNoH may be understood similarly to T-ThrNoH. And `getfield` typing schemes may be understood similarly to `putfield` typing schemes.

(T-ThrNoH)
$$\frac{\begin{array}{l} B(i) = \texttt{throw} \\ \mathsf{Handler}(i)\uparrow \\ \mathcal{A}_i \sqcup \mathcal{S}_i(0) \sqsubseteq \kappa_r \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{err_i} \\ \mathcal{V}_i \leq \mathcal{V}_{err_i} \\ \mathcal{T}_i \leq \mathcal{T}_{err_i} \end{array}}{\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T}, i \rhd B}$$

(T-ThrH)
$$\frac{\begin{array}{l} B(i) = \texttt{throw} \\ \mathsf{Handler}(i) = i' \\ \mathcal{S}_i(0) \sqsubseteq \mathcal{A}_{i'} \\ \mathcal{S}_i(0)\cdot\epsilon \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\ \mathcal{V}_i \leq \mathcal{V}_{i'} \\ \mathcal{T}_i \leq \mathcal{T}_{i'} \\ \forall j \in \mathsf{Dom}(\mathcal{S}_i).\mathcal{A}_i \sqsubseteq \mathcal{S}_i(j) \end{array}}{\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T}, i \rhd B}$$

(T-PtFld)
$$\frac{\begin{array}{l} B(i) = \texttt{putfield}\ f \\ \mathsf{Handler}(i)\downarrow \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \langle R_1, l_1\rangle \cdot \langle R, l\rangle \cdot \mathcal{S}_{i+1} \\ l_1 \sqcup l \sqsubseteq \bigsqcap_{a\in R} \mathcal{T}_i(a,f) \\ \textit{for all } a \in R.\mathcal{T}_{i+1}(a,f) = \langle R_3, l_2\rangle, \\ \qquad\qquad \textit{where } \mathcal{T}_i(a,f) = \langle \_, l_2\rangle \wedge R_1 \subseteq R_3 \\ \mathcal{A}_i \sqsubseteq l, l_1 \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \backslash \{(a,f) \mid a \in R\} \leq \mathcal{T}_{i+1} \backslash \{(a,f) \mid a \in R\} \\ \forall k \in \texttt{region}(i).l \sqsubseteq \mathcal{A}_k \end{array}}{\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T}, i \rhd B}$$

(T-PtFldNoH)
$$\frac{\begin{array}{l} B(i) = \texttt{putfield}\ f \\ \mathsf{Handler}(i)\uparrow \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \kappa \cdot \langle R, l\rangle \cdot (\mathcal{S}_{err_i}\backslash 0) \\ \kappa \sqsubseteq \bigsqcap_{a\in R} \mathcal{T}_i(a,f) \\ a = \mathsf{Fresh}(\mathcal{T}_i) \\ \langle \{a\}, l \sqcup \mathcal{A}_i\rangle \sqsubseteq \mathcal{S}_{err_i}(0) \\ l \sqsubseteq \kappa_r \\ \mathcal{A}_i \sqsubseteq \kappa, l \\ \mathcal{V}_i \leq \mathcal{V}_{err_i} \\ \mathcal{T}_i \oplus \{a \mapsto [f_1 : \texttt{ft}_{\mathcal{A}_i}(f_1), \ldots, f_n : \texttt{ft}_{\mathcal{A}_i}(f_n)]\} \leq \mathcal{T}_{err_i} \end{array}}{\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T}, i \rhd B}$$

(T-PtFldH)
$$\frac{\begin{array}{l} B(i) = \texttt{putfield}\ f \\ \mathsf{Handler}(i) = i' \\ \mathcal{S}_i = \kappa' \cdot \langle R, l\rangle \cdot S, \text{ for some } S \\ a = \mathsf{Fresh}(\mathcal{T}_i) \\ \langle \{a\}, l \sqcup \mathcal{A}_i\rangle \cdot \epsilon \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\ \mathcal{V}_i \leq \mathcal{V}_{i'} \\ \mathcal{T}_i \oplus \{a \mapsto [f_1 : \texttt{ft}_{\mathcal{A}_i}(f_1), \ldots, f_n : \texttt{ft}_{\mathcal{A}_i}(f_n)]\} \leq \mathcal{T}_{i'} \\ \forall j \in \mathsf{Dom}(\mathcal{S}_i).\mathcal{A}_i \sqsubseteq \mathcal{S}_i(j) \end{array}}{\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T}, i \rhd B}$$

(T-Ret)
$$\frac{\begin{array}{l} B(i) = \texttt{return} \\ \mathcal{S}_i(0) \sqcup \mathcal{A}_i \sqsubseteq \kappa_r \\ \mathcal{T}_i \leq \mathcal{T}_{p_f} \end{array}}{\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T}, i \rhd B}$$

(T-ErrI)
$$\frac{\begin{array}{l} B(err_i) \\ \mathcal{S}_i(0) \sqcup \mathcal{A}_i \sqsubseteq \kappa_r \\ \mathcal{T}_i \leq \mathcal{T}_{p_f} \end{array}}{\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T}, i \rhd B}$$

(T-GtFld)
$$\frac{\begin{array}{l} B(i) = \texttt{getfield}\ f \\ \mathsf{Handler}(i)\downarrow \\ \mathcal{S}_i(0) = \langle R, l\rangle \\ \kappa = \bigsqcup_{a\in R} \mathcal{T}_i(a,f) \\ l \sqcup \kappa \sqcup \mathcal{A}_i \cdot (\mathcal{S}_i\backslash 0) \leq_{\mathcal{A}_i} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \\ \forall k \in \texttt{region}(i).l \sqsubseteq \mathcal{A}_k \end{array}}{\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T}, i \rhd B}$$

(T-GtFldH)
$$\frac{\begin{array}{l} B(i) = \texttt{getfield}\ f \\ \mathsf{Handler}(i) = i' \\ \mathcal{S}_i = \langle R, l\rangle \cdot S, \text{ for some } S \\ a = \mathsf{Fresh}(\mathcal{T}_i) \\ \langle \{a\}, l \sqcup \mathcal{A}_i\rangle \cdot \epsilon \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\ \mathcal{V}_i \leq \mathcal{V}_{i'} \\ \mathcal{T}_i \oplus \{a \mapsto [f_1 : \texttt{ft}_{\mathcal{A}_i}(f_1), \ldots, f_n : \texttt{ft}_{\mathcal{A}_i}(f_n)]\} \leq \mathcal{T}_{i'} \\ \forall j \in \mathsf{Dom}(\mathcal{S}_i).\mathcal{A}_i \sqsubseteq \mathcal{S}_i(j) \end{array}}{\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T}, i \rhd B}$$

(T-GtFldNoH)
$$\frac{\begin{array}{l} B(i) = \texttt{getfield}\ f \\ \mathsf{Handler}(i)\uparrow \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \langle R, l\rangle \cdot (\mathcal{S}_{err_i}\backslash 0) \\ \mathcal{S}_i(0) = \langle R, l\rangle \\ \kappa = \bigsqcup_{a\in R} \mathcal{T}_i(a,f) \\ a = \mathsf{Fresh}(\mathcal{T}_i) \\ \langle \{a\}, l \sqcup \mathcal{A}_i\rangle \sqsubseteq \mathcal{S}_{err_i}(0) \\ \mathcal{A}_i \sqsubseteq l \sqsubseteq \kappa_r \\ \mathcal{V}_i \leq \mathcal{V}_{err_i} \\ \mathcal{T}_i \oplus \{a \mapsto [f_1 : \texttt{ft}_{\mathcal{A}_i}(f_1), \ldots, f_n : \texttt{ft}_{\mathcal{A}_i}(f_n)]\} \leq \mathcal{T}_{err_i} \end{array}}{\mathcal{V},\mathcal{S},\mathcal{A},\mathcal{T}, i \rhd B}$$

Fig. 3.11. Typing Schemes for Exception Handling

Example 3.19. Let us return to Example 3.18 above. As already mentioned, this program is interferent if y and f_1 are `H`, x and f_2 are `L` and the exception table is $(1, 7, 8)$ (meaning the handler for a **Throwable** exception commencing at instruction 8 applies to instruction range 1 to 7). In a normal execution field f_2 is assigned the value v_2, while in an exceptional case (y contains a null reference) instructions 4, 5 and 6 are never reached. Since f_2 is low, it is observable and hence there is an implicit flow from f_1 to f_2. This program is *rejected* by JVM^s_E. First note that $\mathsf{region}(3) = \{4, 5, 6, 7\}$ and $\mathsf{region}(6) = \{7\}$. Then the security environment of the instructions in $\{4, 5, 6, 7\}$ is high and the type system prevents the assignment to low fields in high contexts, i.e. the `putfield` at line 6 is rejected.

In the case that the handler is not defined and κ_r is `L`, then this program is interferent since the exception that is raised is high (the object y is high). The type system *rejects* this program by requiring that the object which throws an exception has at least level κ_r.

Example 3.20. The program of Example 3.17 also is rejected by JVM^s_E. We note that the security environment of the instructions in $\{4, 5, 6, 9, 10, 11, 12\}$ is high ($\mathsf{region}(3) = \{4, 5, 6, 9, 10, 11, 12\}$) and the type system prevents the assignment to fields of low objects in high contexts, i.e. the `putfield` at line 11 is rejected.

3.5 JVM^s_C - Adding Method Calls to JVM^s_E

This section presents our final extension to JVM^s, namely JVM^s_C, which results from adding *method calls*. We briefly explain the JVM method call mechanism and extend the operational semantics and inference schemes for JVM^s_C. The following table summarizes our three extensions:

System	Description
JVM^s	Core bytecode
JVM^s_E	Core bytecode with exceptions
JVM^s_C	Core bytecode with exceptions and method calls

3.5.1 JVM Method Calls

A method invocation completes normally or abruptly. A method invocation completes normally if that invocation does not cause an exception to be thrown, either directly from the Java virtual machine or as a result of executing an explicit throw statement. If the invocation of the current method completes normally, then a value may be returned to the invoking method. This occurs when the invoked method executes one of the return instructions, the choice of which must be appropriate for the type of the value being returned (if any). Execution then continues normally in the invoking method's frame with the returned value (if any) pushed onto the operand stack of that frame. A method invocation completes abruptly if execution of a Java virtual machine instruction within the method causes the Java virtual machine to throw an exception, and that exception is not handled within the method. Execution of a throw instruction also causes an exception to be explicitly thrown and, if the exception is not caught by the current method, results in abrupt method invocation completion. A method invocation that completes abruptly never returns a value to its invoker [75].

3.5.2 JVM^s_C

In the presence of methods and exceptions, information leakage can be caused by propagation of exceptions to caller methods. Furthermore, other kind of leakage that arises with the extension to methods is caused by dynamic method dispatch, i.e. when classes contain different implementations of the same method, and different methods codes are executed depending on the dynamic type of their target object. We will illustrate these cases with examples. After doing so we introduce the syntax of JVM^s_C.

We recall from that we have a set $\mathbb{C}$ of class names that is as a hierarchy to model the inheritance of classes. Each class comes equipped with set $\mathbb{M}$ of method names; letters m, m' range over $\mathbb{M}$. Each method m possesses its local variables set $\mathbb{X}_m$ and its list of instructions B_m. The set of instructions of JVM^s_C results from extending those of JVM^s_E the new instruction `invokevirtual` m where m is any method name.

Example 3.21. A example of information leakage due to exceptions. Consider the method m, in Figure 3.12, with exception handler $(5, 7, 9)$, method m_3 with exception handler $(3, 3, 5)$ and method m_2 without any handler. The local variable $y_{\mathtt{H}}$ is passed as parameter to method m_2. If the value of $y_{\mathtt{H}}$ at run-time is *null* then an exception is thrown and handled in m. The exception handler specifies an assignment to a low field $f'_{\mathtt{L}}$. An attacker, who can observe the value of $f'_{\mathtt{L}}$, will detect if variable $y_{\mathtt{H}}$ has value *null* or not according to the value of $f'_{\mathtt{L}}$ after execution of method m. The attacker can also infer information about $z_{\mathtt{H}}$ since the exception may be handled by the handler method m_3, assigning low field $f'_{\mathtt{L}}$, depending on the value of $z_{\mathtt{H}}$.

Code of m	Code of m_2	Code of m_3
1. `load` $y_{\mathtt{H}}$	1. `load` $y_{\mathtt{H}}$	1. `load` $y_{\mathtt{H}}$
2. `load` $y_{\mathtt{H}}$	2. `load` *this*	2. `load` *this*
3. `load` $z_{\mathtt{H}}$	3. `putfield` $f_{\mathtt{H}}$	3. `putfield` $f_{\mathtt{H}}$
4. `if` 7	4. `push` 1	4. `goto` 7
5. `invokevirtual` m_2	5. `return`	5. `push` 0
6. `goto` 12		6. `putfield` $f''_{\mathtt{L}}$
7. `invokevirtual` m_3		7. `push` 1
8. `goto` 12		8. `return`
9. `push` 1		
10 `putfield` $f'_{\mathtt{L}}$		
11 `push` 1		
12. `return`		

Fig. 3.12. Information leak due to exceptions.

Example 3.22. These examples (taken from [94]) illustrate how object-oriented features can lead to interference. Let class C be a superclass of a class D. Let method foo be declared in D, and a method m declared in C and overridden in D as illustrated by the source program in Figure 3.13. At run-time, either code $C.m$ or code $D.m$ is executed depending on the value of high variable $y_{\mathtt{H}}$. Information about $y_{\mathtt{H}}$ may be inferred by observing the return value of method m.

Operational semantics of JVM^s_C. While JVM states contain a frame stack to handle method invocations, it is convenient (as discussed in [94]) for showing the correctness of static analysis to rely on an equivalent semantics where method invocation is performed in one go. While small-step semantics uses a call stack to store the calling context and retrieve it during a return instruction, method invocation directly executes the full evaluation of the called method from an initial state to a return value and uses it to continue the current computation. The resulting semantics is termed mixed-step semantics.

This choice simplifies the notion of indistinguishability between states. In the presence of a small-step semantics states possess stack of frames (one frame corresponding to each method in the calling chain) and hence indistiguishability must take account of frames of high and low methods which can throw and propagate low and high exceptions. It is also needed for indistiguishability to state if the method is invoked in a low or high target object.

The reduction rules defining the mixed-step operational semantics is given in Figure 3.14. A method identifier may correspond to several methods in the class hierarchy according to overriding of methods.

```
class C{
      int m(){return 0; }
}
class D extends C{
      int m(){ return 1; }
      int foo(boolean y){
         return ((y? new C() : this).m());
      }
}
```

(a) Source Code

```
D.foo :
    1. load y_H
    2. if 5                    C.m :            D.m :
    3. new C                      1. push 0        1. push 1
    4. goto 6                     2. return        2. return
    5. load this
    6. invokevirtual m
    4. return
```

(b) Bytecode

Fig. 3.13. Information leak by object-oriented features.

But, we assume, as in the JVM, that the operational semantics of `invokevirtual` is deterministic, and given by a function lookup that takes a method identifier and a class name (that corresponds to the dynamic type of the target object of the method) given by the function $\mathsf{dynamic}$ and returns the body of the method to be executed. The function $\mathsf{Handler}$ is now parameterized by the subscript B, and it selects the appropriate handler for a given program point. If a handler does not exist inside B, then $\mathsf{Handler}_B$ is undefined. The operational semantics of the remaining instructions is adapted from the previous sections in the obvious way.

$$\frac{\begin{array}{c} B(i) = \texttt{invokevirtual}\ m \quad o' = \mathsf{Fresh}(\eta) \quad B' = \mathsf{lookup}(m, \mathsf{dynamic}(o)) \\ \langle 1, \{this \mapsto o, \overrightarrow{x \mapsto v_1}\}, \epsilon, \eta\rangle \longrightarrow^*_{B'} \langle p'_f, \langle o'\rangle, \eta'\rangle \quad \mathsf{Handler}_B(i) = i' \end{array}}{\langle i, \alpha, \overrightarrow{v_1} \cdot o \cdot \sigma, \eta\rangle \longrightarrow_B \langle i', \alpha, \langle o'\rangle \cdot \epsilon, \eta'\rangle}\ \text{(R-InvkVrtlH)}$$

$$\frac{\begin{array}{c} B(i) = \texttt{invokevirtual}\ m \quad o' = \mathsf{Fresh}(\eta) \quad \mathsf{Handler}_B(i)\uparrow \quad B' = \mathsf{lookup}(m, \mathsf{dynamic}(o)) \\ \langle 1, \{this \mapsto o, \overrightarrow{x \mapsto v_1}\}, \epsilon, \eta\rangle \longrightarrow^*_{B'} \langle p'_f, \langle o'\rangle, \eta'\rangle \end{array}}{\langle i, \alpha, \overrightarrow{v_1} \cdot o \cdot \sigma, \eta\rangle \longrightarrow_B \langle err_i, \alpha, \langle o'\rangle \cdot \sigma, \eta'\rangle}\ \text{(R-InvkVrtlNoH)}$$

$$\frac{\begin{array}{c} B(i) = \texttt{invokevirtual}\ m \quad o \in \mathsf{Dom}(\eta) \quad B' = \mathsf{lookup}(m, \mathsf{dynamic}(o)) \\ \langle 1, \{this \mapsto o, \overrightarrow{x \mapsto v_1}\}, \epsilon, \eta\rangle \longrightarrow^*_{B'} \langle p'_f, v, \eta'\rangle \end{array}}{\langle i, \alpha, \overrightarrow{v_1} \cdot o \cdot \sigma, \eta\rangle \longrightarrow_B \langle i+1, \alpha, v \cdot \sigma, \eta'\rangle}\ \text{(R-InvkVrtl)}$$

Fig. 3.14. Additional Reduction Schemes for Method Calls

3.5.3 Type System

As mentioned earlier, in the presence of methods and exceptions, information leakage can be caused by propagation of exceptions to the calling method. This kind of leakage is prevented by providing security signatures that impose constraints on parameters, the result, assignments and exceptions that must be satisfied for methods to be typable. Signatures for the called methods are used in the typing rules for calling methods, achieving modularity in this way. Thus, the analysis assumes that methods come equipped with a security signature of the form

$$\mathcal{A}_1, \mathcal{T}_1, \mathcal{T}_{p_f} \rhd ((\overrightarrow{x : \kappa}, \kappa_r, E), B)$$

where $\overrightarrow{x : \kappa} = x_1 : \kappa_1, \ldots, x_n : \kappa_n$ and $\kappa_1, \ldots, \kappa_n$ are the security labels of the formal parameters $x_1, \ldots, x_n$. κ_r is the security label of the value returned by the method and is the maximal security level of the exceptions not handled in the method might be thrown. If the exceptional effect of a method m is low ($\kappa_r = \mathtt{L}$), then if an exception is propagated from m, it will correspond to a low exception. A low exception is an exception thrown in a low environment (possibly under a low conditional) and caused by a low expression, e.g. an access to field $x.f$ where x is a low variable holding a null value. $\mathcal{A}_1$ is the context level in which the method will be executed (the level of the first instruction). $\mathcal{T}_1$ is the initial heap and $\mathcal{T}_{p_f}$ is the final heap. The set E contains the throwable class names of exceptions not handled in the method. If the method handles all its exceptions then $E = \emptyset$. We assume that every variable appearing in the method is included in the signature of the method, i.e. the frame variables $\mathcal{V}_1$ is defined by the parameters $(\overrightarrow{x : \kappa})$.

A method could have more than one defined security signature given that it could behave differently with different target objects. Then security signatures are therefore given by a function mt. This function is parameterized by a method name and a security label, $\mathsf{mt}_m(\kappa)$. κ is the supremum from the security label of the target object and the security context where the method is invoked. The symbolic locations in κ help determine which method is invoked.

Before addressing the typing rules for JVM^s_C, we revisit the notion of control dependence regions.

Control Dependence Regions for JVM^s_C. Given that method invocation is a form of conditional jump (in case that an exception not handled in the method might be thrown) control dependence regions must be extended. The set of conditional control points is modified as follows:

$$\begin{aligned}\mathsf{Dom}(B)^\sharp = \{k \in \mathsf{Dom}(B) \mid\ & B(k) = \mathtt{if}\ i \text{ or } B(k) = \mathtt{putfield}\ f \text{ or } B(k) = \mathtt{getfield}\ f \text{ or} \\ & (B(k) = \mathtt{invokevirtual}\ m \text{ and } (\exists l.\mathsf{mt}_m(l) = \mathcal{A}_1, \mathcal{T}_1, \mathcal{T}_{p_f} \rhd ((\overrightarrow{x : \kappa}, \kappa_r, E), B') \text{ and } E \neq \emptyset))\}.\end{aligned}$$

Then we extend the successor relation (Def. 3.9) to include the possible change of control flow:

- If $B(i) = \mathtt{invokevirtual}\ m$ then $i \mapsto i+1$ and if $\mathsf{Handler}_B(i) = i'$, then $i \mapsto i'$ otherwise $i \mapsto i_{err}$.

The SOAP properties, however, are not modified since that notion is already parametrized by $\mathsf{Dom}(B)^\sharp$.

Typability

The additional typing schemes are presented in Fig. 3.15. There are three typing schemes for `invokevirtual`. The T-InvkVrtl and T-InvkVrtlH schemes types a `invokevirtual` at a program point that has a handler. The T-InvkVrtl and T-InvkVrtlNoH are applied if the program point has a no handler. We now describe each of these in turn. The security level in which the method is executed is given by the target object level and the context level $(\mathcal{A}_i \sqcup \kappa)$. The security level of the return value is $\kappa \sqcup \kappa'_r \sqcup \mathcal{A}_i$. The level κ prevents flows due to execution of two distinct methods. A high target object may mean that the body of the method to be executed is different in two executions. The security level κ_r prevents that the method result flows to variables or fields with lower level and $\mathcal{A}_i$ prevents implicit flows. The constraints on the operand stack are due to the fact that the first elements are removed (parameters and object invoker) from the operand stack and the return value is pushed on the stack. Furthermore, the security level in which the method is executed ($\mathcal{A}_i \sqcup \kappa$) is to avoid invocation of methods with low effect on the heap with high target objects or in high contexts. Finally, if the invoked method does not catch all exceptions then the method call instruction is a branching point. Hence, the security level of $\mathsf{region}(i)$ must be greater than or equal to the security level of the return value.

We assume an implicit function mapping the arguments to parameters of B' and transposing flows between parameters in B' in flows between mapped values in B. It is used especially in heap constraint $\mathcal{T}_i \leq \mathcal{T}'_1 \leq \mathcal{T}'_{pf} \leq \mathcal{T}_{err_i}$.

The second scheme is applied in case of abnormal termination of the invoked method and a handler is defined for the current instruction. The main requirements in this scheme are $\mathcal{S}_{i'} = \kappa'_r \sqcup \mathcal{A}_i \cdot \epsilon$ and $\forall j \in \mathsf{Dom}(\mathcal{S}_i). \kappa \sqcup \mathcal{A}_i \sqcup \kappa'_r \sqsubseteq \mathcal{S}_i(j)$. These constraints are necessary since upon raising an exception, the contents of the operand stack is cleared and the security level of the exception handler is pushed onto the top of stack. If exception is high, then low values on the stack cannot be removed.

The last one is applied in case of abnormal termination of the invoked method and no handler is defined for the current instruction (an exception escape from the current method). The called method terminates by a uncaught exception and then propagates the thrown exception. Therefore the security level of the exception is push onto the top of the stack of the err_i instruction ($\kappa'_r \sqcup \mathcal{A}_i \sqsubseteq \mathcal{S}_{err_i}(0)$). Finally, the security level of return object must be $\mathcal{A}_i \sqcup \kappa \sqcup \kappa'_r \sqsubseteq \kappa_r$.

$$\text{(T-InvkVrtl)}\quad \frac{\begin{array}{l} B(i) = \texttt{invokevirtual}\ m \\ \mathsf{Handler}_B(i)\downarrow \\ \mathsf{mt}_m(\kappa \sqcup \mathcal{A}_i) = \mathcal{A}'_1, \mathcal{T}'_1, \mathcal{T}'_{pf} \rhd ((\overrightarrow{x' : \kappa'}, \kappa'_r, E'), B') \\ \mathcal{A}_i \sqcup \kappa = \mathcal{A}'_1 \\ \mathcal{S}_i = \kappa'_1 \cdots \kappa'_n \cdot \kappa \cdot S,\ \text{for some}\ S \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \kappa'_1 \cdots \kappa'_n \cdot \kappa \cdot (\mathcal{S}_{i+1} \backslash 0) \\ \mathcal{A}_i \sqsubseteq \kappa'_1, \cdots, \kappa'_n, \kappa \\ \kappa \sqcup \kappa'_r \sqcup \mathcal{A}_i \sqsubseteq \mathcal{S}_{i+1}(0) \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}'_1 \leq \mathcal{T}'_{pf} \leq \mathcal{T}_{i+1} \\ \forall k \in \texttt{region}(i). \kappa \sqcup \kappa'_r \sqsubseteq \mathcal{A}_k,\ \text{if}\ i \in \mathsf{Dom}(B^\sharp) \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B}$$

$$\text{(T-InvkVrtlH)}\quad \frac{\begin{array}{l} B(i) = \texttt{invokevirtual}\ m \\ \mathsf{Handler}_B(i) = i' \\ \mathsf{mt}_m(\kappa \sqcup \mathcal{A}_i) = \mathcal{A}'_1, \mathcal{T}'_1, \mathcal{T}'_{pf} \rhd ((\overrightarrow{x' : \kappa'}, \kappa'_r, E'), B') \\ E' \neq \emptyset \\ \mathcal{A}_i \sqcup \kappa = \mathcal{A}'(1) \\ \kappa'_r \sqcup \mathcal{A}_i \cdot \epsilon \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\ \mathcal{T}_i \leq \mathcal{T}'_1 \leq \mathcal{T}'_{pf} \leq \mathcal{T}_{i'} \\ \mathcal{V}_i \leq \mathcal{V}_{i'} \\ \forall j \in \mathsf{Dom}(\mathcal{S}_i). \mathcal{A}_i \sqsubseteq \mathcal{S}_i(j) \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B}$$

$$\text{(T-InvkVrtlNoH)}\quad \frac{\begin{array}{l} B(i) = \texttt{invokevirtual}\ m \\ \mathsf{Handler}_B(i)\uparrow \\ \mathsf{mt}_m(\kappa \sqcup \mathcal{A}_i) = \mathcal{A}'_1, \mathcal{T}'_1, \mathcal{T}'_{pf} \rhd ((\overrightarrow{x' : \kappa'}, \kappa'_r, E'), B') \\ E' \neq \emptyset \\ \mathcal{A}_i \sqcup \kappa = \mathcal{A}'(1) \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \kappa'_1 \cdots \kappa'_n \cdot \kappa \cdot (\mathcal{S}_{err_i} \backslash 0) \\ \kappa'_r \sqcup \mathcal{A}_i \sqsubseteq \mathcal{S}_{err_i}(0) \\ \mathcal{A}_i \sqcup \kappa \sqcup \kappa'_r \sqsubseteq \kappa_r \\ \mathcal{V}_i \leq \mathcal{V}_{err_i} \\ \mathcal{T}_i \leq \mathcal{T}'_1 \leq \mathcal{T}'_{pf} \leq \mathcal{T}_{err_i} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B}$$

Fig. 3.15. Typing Schemes for Method Calls

Definition 3.23 (Well-Typed Method). *A method M is well-typed if for all security signatures of M, $\mathsf{mt}_M(l) = l, \mathcal{T}_1, \mathcal{T}_{p_f} \rhd ((\overrightarrow{x : \kappa}, \kappa_r, E), B)$, there exists a typing context $(\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T})$, where*

$\mathcal{V} = \{\overrightarrow{x \mapsto \kappa}, \mathcal{V}_2, \cdots, \mathcal{V}_{p_f}\}$*, for some* $\mathcal{V}_2, \cdots, \mathcal{V}_{p_f}$*;*
$\mathcal{S} = \{\epsilon, \mathcal{S}_2, \cdots, \mathcal{S}_{p_f}\}$*, for some* $\mathcal{S}_2, \cdots, \mathcal{S}_{p_f}$*;*
$\mathcal{A} = \{l, \mathcal{A}_2, \cdots, \mathcal{A}_{p_f}\}$*, for some* $\mathcal{A}_2, \cdots, \mathcal{A}_{p_f}$*;*
$\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_{p_f}\}$*, for some* $\mathcal{T}_2, \cdots, \mathcal{T}_{p_f-1}$*,*

such that the type judgement $\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T} \rhd M$ holds.

A program is well-typed if all its methods are.

Definition 3.24 (Well-Typed Program). *A program P is well-typed if for each declared method m and for each security signature of m, $\mathsf{mt}_m(l) = l, \mathcal{T}_1, \mathcal{T}_{p_f} \rhd ((\overrightarrow{x : \kappa}, \kappa_r, E), B)$, there exists a typing context $(\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T})$, where*

$\mathcal{V} = \{\overrightarrow{x \mapsto \kappa}, \mathcal{V}_2, \cdots, \mathcal{V}_{p_f}\}$*, for some* $\mathcal{V}_2, \cdots, \mathcal{V}_{p_f}$*;*
$\mathcal{S} = \{\epsilon, \mathcal{S}_2, \cdots, \mathcal{S}_{p_f}\}$*, for some* $\mathcal{S}_2, \cdots, \mathcal{S}_{p_f}$*;*
$\mathcal{A} = \{l, \mathcal{A}_2, \cdots, \mathcal{A}_{p_f}\}$*, for some* $\mathcal{A}_2, \cdots, \mathcal{A}_{p_f}$*;*
$\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_{p_f}\}$*, for some* $\mathcal{T}_2, \cdots, \mathcal{T}_{p_f-1}$*,*

such that the type judgement $\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T} \rhd m$ holds.

Let us return to Example 3.21 and Example 3.22. In the former if the value of $y_{\mathtt{H}}$ at run-time is *null* then an exception is thrown and handled in m. The exception handler specifies an assignment to a low field $f'_{\mathtt{L}}$. An attacker, who can observe the value of $f'_{\mathtt{L}}$, will detect if variable $y_{\mathtt{H}}$ has value *null* or not according to the value of $f'_{\mathtt{L}}$ after execution of method m. The attacker can also infer information about $z_{\mathtt{H}}$ since the exception may be handled by the handler method m_3, assigning low field $f'_{\mathtt{L}}$, depending on the value of $z_{\mathtt{H}}$. This kind of programs is *rejected* by the type system, since the type system disallows invocation of methods with high exceptions effect from methods with low value return. Furthermore, the exception not handled by m_2 is high then the lift to high of the region of `invokevirtual` m_2 in the typing rule disallows the assignment to any low fields in its region.

The second example we recall that at run-time, either code $C.m$ or code $D.m$ is executed depending on the value of high variable $y_{\mathtt{H}}$. Information about $y_{\mathtt{H}}$ may be inferred by observing the return value of method m. The type system *prevents* this flow by requiring that the return value level is determined by $\kappa \sqcup \kappa'_r \sqcup \mathcal{A}_i$, where κ is the target object security level.

3.6 Noninterference

This section proves the absence of illicit information flows by means of *noninterference* [62]. A notion of equivalence of machine states relative to a security level is introduced, the intuition being that two related states are *indistinguishable* for an external observer. In essence, noninterference ensures that any two computations wich are fired from initial states that differ only in values indistinguishable for an external observer, are equivalent. The principle effort in proving this result is in obtaining an appropriate notion of indistinguishable states. This of course depends on the types of illicit flows that are to be captured. In particular, we have to provide a definition that caters for the features advertised in the introduction, including unrestricted reuse of local variables and a more precise control of the operand stack.

We begin our discussion with some preliminary notions. Given that in any two runs of a program different locations may be allocated, it is convenient to introduce a (partial) bijection β between locations [19]. For example, two runs of `new` C could allocate different locations, say o_1 and o_2, in the heap to the new object. This bijection helps establish that these locations are related by setting $\beta(o_1) = o_2$.

Given that JVM^s tracks the types of fields via symbolic locations we also introduce a pair of (partial) bijections between symbolic locations and locations themselves: $(\beta^{\lhd}, \beta^{\rhd})$ (cf. Figure 3.16). In our example, $\beta^{\lhd}(a) = o_1$ and $\beta^{\rhd}(a) = o_2$. Notice that both locations get assigned the same symbolic location given that it arises from the typing derivation of `new` C. The sum of β and $(\beta^{\lhd}, \beta^{\rhd})$ is called a *location bijection set*. Location bijection sets shall thus be required for relating heaps (as explained below). We write f^{-1} for the inverse of function f and $f \subseteq f'$ for the inclusion of functions. A function f is included in f' if $\mathsf{Dom}(f) \subseteq \mathsf{Dom}(f')$, $\mathsf{Ran}(f) \subseteq \mathsf{Ran}(f')$ and for all $(a, b) \in f$ then $(a, b) \in f'$.

Definition 3.25 (Location Bijection Set). *A location bijection set β consists of (1) a (partial) bijection β_{loc} between locations; and (2) a pair of (partial) bijections $(\beta^{\lhd}, \beta^{\rhd})$ between symbolic locations and locations with*

1. $\mathsf{Dom}(\beta_{loc}) \subseteq \mathsf{Ran}(\beta^{\lhd})$.
2. $\mathsf{Ran}(\beta_{loc}) \subseteq \mathsf{Ran}(\beta^{\rhd})$.

s.t. for all $o \in \mathsf{Dom}(\beta_{loc})$, $\beta^{\lhd^{-1}}(o) = \beta^{\rhd^{-1}}(\beta_{loc}(o)))$.

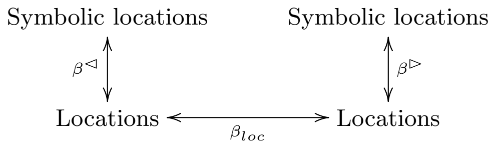

$$\begin{tikzcd}
\text{Symbolic locations} \arrow[d, leftrightarrow, "\beta^{\lhd}"'] & \text{Symbolic locations} \arrow[d, leftrightarrow, "\beta^{\rhd}"] \\
\text{Locations} \arrow[r, leftrightarrow, "\beta_{loc}"'] & \text{Locations}
\end{tikzcd}$$

Fig. 3.16. Location bijection Set

By default, we write $\beta(o)$ to mean $\beta_{loc}(o)$.

Definition 3.26. *We define location bijection set β' to be an extension of β, written $\beta \subseteq \beta'$, if $\beta_{loc} \subseteq \beta'_{loc}$ and $\mathsf{Ran}(\beta^{\lhd}) \subseteq \mathsf{Ran}(\beta'^{\lhd})$ and $\mathsf{Ran}(\beta^{\rhd}) \subseteq \mathsf{Ran}(\beta'^{\rhd})$.*

Definition 3.27. *We write β^{id} for a location bijection set β such that $\beta^{\mathsf{id}}_{loc}$ is the identity on locations (identity over the domain of β_{loc}) and $\beta^{\mathsf{id}\lhd} = \beta^{\lhd}$, $\beta^{\mathsf{id}\rhd} = \beta^{\lhd}$.*

Definition 3.28. *Finally, define the inverse of a location bijection set β, denoted $\hat{\beta}$, to be: $\hat{\beta}_{loc} = \beta^{-1}_{loc}$ and $\hat{\beta}^{\lhd} = \beta^{\rhd}$ and $\hat{\beta}^{\rhd} = \beta^{\lhd}$.*

A small set of requirements state some basic properties on how the heaps and heaps types are related by a location bijection set.

Property 3.29 (Well-Defined Location Bijection Set). Let η_1, η_2 be heaps, T_1, T_2 heap types then a location bijection set β is well-defined w.r.t. η_1, η_2, T_1 and T_2 if

1. $\mathsf{Dom}(\beta_{loc}) \subseteq \mathsf{Dom}(\eta_1)$ and $\mathsf{Ran}(\beta_{loc}) \subseteq \mathsf{Dom}(\eta_2)$,
2. $\mathsf{Ran}(\beta^{\lhd}) \subseteq \mathsf{Dom}(\eta_1)$ and $\mathsf{Ran}(\beta^{\rhd}) \subseteq \mathsf{Dom}(\eta_2)$,
3. $\mathsf{Dom}(\beta^{\lhd}) \subseteq \mathsf{Dom}(T_1)$ and $\mathsf{Dom}(\beta^{\rhd}) \subseteq \mathsf{Dom}(T_2)$,
4. for every $o \in \mathsf{Dom}(\beta_{loc})$, $\mathsf{Dom}(\eta_1(o)) = \mathsf{Dom}(\eta_2(\beta_{loc}(o)))$.

3.6.1 Indistinguishability - Definitions

In order to relate states we need to be able to relate each of its components. This includes values, variable arrays, stacks and heaps, all of which are addressed in this section. *Indistinguishability* for states at some security level λ is formulated by considering each of its components at a time. States are declared indistinguishable depending on what may be observed and this, in turn, depends on the security level l of the observer.

Values. The case for values is standard: public values must be identical (except for the case of locations where they have to be related via the location bijection β) for them to be indistinguishable, however any pair of secret values are indistinguishable regardless of their form.

Definition 3.30 (Value Indistinguishability). *Given values v_1, v_2, security levels l, λ and β a location bijection set we define $\beta, l \vdash v_1 \sim^\lambda v_2$ ("values v_1 and v_2 are indistinguishable at level λ w.r.t. observer level l") as follows:*

$$\text{(L-NULL)}\quad \frac{l \sqsubseteq \lambda}{\beta, l \vdash \mathit{null} \sim^\lambda \mathit{null}} \qquad \text{(L-INT)}\quad \frac{v \in \mathbb{Z} \qquad l \sqsubseteq \lambda}{\beta, l \vdash v \sim^\lambda v}$$

$$\text{(L-LOC)}\quad \frac{v_1, v_2 \in \mathbb{L} \quad l \sqsubseteq \lambda \quad \beta(v_1) = v_2}{\beta, l \vdash v_1 \sim^\lambda v_2} \qquad \text{(H-VAL)}\quad \frac{l \not\sqsubseteq \lambda}{\beta, l \vdash v_1 \sim^\lambda v_2}$$

Local Variable Arrays. In the case that the current security level of the program counter is low, both program runs are assumed to be executing the same instructions, in lock-step. We thus compare pointwise the value of each variable using the security label indicated by $\mathcal{V}_i$, the variable array type at instruction i. Note that this does not preclude local variable reuse: if one computation reuses a register, given that both are executing the same instruction, so will the other computation; moreover, both values shall be related assuming the initial states are related.

However, if the current security level of the program counter is high, then computations may proceed independently before converging at a junction point. Meanwhile, reuse of a local variable, say x, could take place by writing a high level value where a previously low level one was held. By comparing the values of x in both computations using the join of their levels we can assure that if reuse takes place, then the contents of the variables are not compared (given that these contents are no longer related). Note that once the junction point of the high level region, in which these computations occur, is reached the subtyping conditions on heap types shall ensure that from that point on, the previously low level variable should in fact be considered no longer inspectable (i.e. should be considered high).

Definition 3.31 (Local Variable Array Indistinguishability). *Let α_1, α_2 be local variable arrays, V_1, V_2 frame types, β a location bijection set and l, λ a security levels. We write $\beta, (V_1, V_2), l \vdash \alpha_1 \sim^\lambda \alpha_2$ (or $\beta, V_1, l \vdash \alpha_1 \sim^\lambda \alpha_2$ when $V_1 = V_2$ and $l \sqsubseteq \lambda$).*

- *Low-indist. Local Variable ($l \sqsubseteq \lambda$ and $V_1 = V_2$). α_1 and α_2 are considered low-indist. at level λ if for all $x \in \mathbb{X}$,*
$$\beta, \lfloor V_1(x) \rfloor \vdash \alpha_1(x) \sim^\lambda \alpha_2(x).$$
- *High-indist. Local Variable ($l \not\sqsubseteq \lambda$). α_1 and α_2 are considered high-indist. at level λ if for all $x \in \mathbb{X}$,*
$$\beta, \lfloor V_1(x) \sqcup V_2(x) \rfloor \vdash \alpha_1(x) \sim^\lambda \alpha_2(x).$$

Heaps. In order to compare heaps η_1 and η_2 we verify that all objects allocated in η_1 are indistinguishable with regards to their corresponding objects, as dictated by β, in the other heap (cf. condition 5 of Def.3.32). Before comparing objects it is determined whether references o allocated in η_1 and which have a related allocated reference $\beta(o)$ in η_2 are indeed of the same type (cf. condition 4 of Def.3.32). Then, they are compared field by field using the security level given by the heap type at the relevant program point (if the instruction is i, then it would be $\mathcal{T}_i(\beta^{\triangleleft-1}(o), f)$, see Def. 3.34). The remaining conditions in the definition below are sanity checks that are in fact preserved by reduction. Regarding the use of $T_1(\beta^{\triangleleft-1}(o), f)$ in condition 5, rather than $T_2(\beta^{\triangleright-1}(\beta(o)), f)$, the reader should note that the following invariant is seen to hold all along the execution of the program: for every $o \in \mathsf{Dom}(\beta)$, for every $f \in \mathsf{Dom}(o)$, $\lfloor T_1(\beta^{\triangleleft-1}(o), f)\rfloor = \lfloor T_2(\beta^{\triangleright-1}(\beta(o)), f)\rfloor$. I.e. the field security level is fixed along the execution of the program.

Definition 3.32 (Heap Indistinguishability). *Let η_1, η_2 be heaps, T_1, T_2 heap types, β a location bijection set and λ security level. We define η_1 and η_2 to be indistinguishable at T_1, T_2 under β (we write $\beta, (T_1, T_2) \vdash \eta_1 \sim^\lambda \eta_2$ or $\beta, T_1 \vdash \eta_1 \sim^\lambda \eta_2$ in the case that $T_1 = T_2$) if:*

1. *β is well-defined w.r.t. η_1, η_2, T_1 and T_2, and*
2. *for every $o \in \mathsf{Dom}(\beta)$, for every field name $f \in \mathsf{Dom}(\eta_1(o))$*

$$\beta, \lfloor T_1(\beta^{\triangleleft-1}(o), f)\rfloor \vdash \eta_1(o, f) \sim^\lambda \eta_2(\beta(o), f).$$

Stacks. Assuming that computation takes place in lock-step, with a program counter set to low, their respective stacks will have the same size and its elements will be related pointwise. This intuition is formalised in the notion of low-indistinguishable stacks below. In the case that a conditional is executed and that the topmost element on the stack is high, computations may progress independently, as mentioned above. As a consequence, their corresponding stacks may vary in the number and types of the additional elements, although all elements added in the region of the conditional shall of course have a high security level. However, at all times both stacks shall contain a common substack, namely those that were related just before executing the conditional.

Definition 3.33 (Stack Indistinguishability). *Let σ_1, σ_2 be stacks, S_1, S_2 stack types, β a location bijection set and l, λ security levels. We write $\beta, (S_1, S_2), l \vdash \sigma_1 \sim^\lambda \sigma_2$ (or $\beta, S_1, l \vdash \sigma_1 \sim^\lambda \sigma_2$, when $S_1 = S_2$ and $l \sqsubseteq \lambda$).*

- *Low-indist. stacks ($l \sqsubseteq \lambda$).*

$$\text{(L-NIL)}\ \frac{l \sqsubseteq \lambda}{\beta, \epsilon, l \vdash \epsilon \sim^\lambda \epsilon} \qquad \text{(CONS-L)}\ \frac{\beta, S, l \vdash \sigma_1 \sim^\lambda \sigma_2 \quad \beta, \lfloor\kappa\rfloor \vdash v_1 \sim^\lambda v_2 \quad l \sqsubseteq \lambda}{\beta, \kappa \cdot S, l \vdash v_1 \cdot \sigma_1 \sim^\lambda v_2 \cdot \sigma_2}$$

- *High-indist. stacks ($l \not\sqsubseteq \lambda$).*

$$\text{(H-LOW)}\ \frac{\beta, S, l \vdash \sigma_1 \sim^\lambda \sigma_2 \quad l \sqsubseteq \lambda \quad l' \not\sqsubseteq \lambda}{\beta, (S, S), l' \vdash \sigma_1 \sim^\lambda \sigma_2}$$

$$\text{(H-CONS-L)}\ \frac{\beta, (S_1, S_2), l \vdash \sigma_1 \sim^\lambda \sigma_2 \quad l, \lfloor\kappa\rfloor \not\sqsubseteq \lambda}{\beta, (\kappa \cdot S_1, S_2), l \vdash v \cdot \sigma_1 \sim^\lambda \sigma_2} \qquad \text{(H-CONS-R)}\ \frac{\beta, (S_1, S_2), l \vdash \sigma_1 \sim^\lambda \sigma_2 \quad l, \lfloor\kappa\rfloor \not\sqsubseteq \lambda}{\beta, (S_1, \kappa \cdot S_2), l \vdash \sigma_1 \sim^\lambda v \cdot \sigma_2}$$

Machine states. Finally, we address indistinguishability of machine states. Two states are indistinguishable if they are either both executing in a high region (in which case we say they are *high-indistinguishable*) or they are both executing the same instruction in a low region (*low-indistinguishable*). Furthermore, in any of these cases the variable array types, operand stacks and heaps should also be indistinguishable. Note that this notion is relative to a current location bijection set.

Definition 3.34 (Machine state indistinguishability). *Given security level λ, β a location bijection set, $\beta \vdash \langle i, \alpha, \sigma, \eta\rangle \sim^\lambda \langle i', \alpha', \sigma', \eta'\rangle$ holds iff*

1. $\mathcal{A}_i, \mathcal{A}_{i'} \not\sqsubseteq \lambda$ *or* $(\mathcal{A}_i = \mathcal{A}_{i'} \sqsubseteq \lambda$ *and* $i = i')$*;*
2. $\beta, (\mathcal{V}_i, \mathcal{V}_{i'}), \mathcal{A}_i \vdash \alpha \sim^\lambda \alpha'$*;*
3. $\beta, (\mathcal{S}_i, \mathcal{S}_{i'}), \mathcal{A}_i \vdash \sigma \sim^\lambda \sigma'$*; and*
4. $\beta, (\mathcal{T}_i, \mathcal{T}_{i'}) \vdash \eta \sim^\lambda \eta'$*.*

Definition 3.35 (Final State Indistinguishability). *Given security level λ, β a location bijection set, $\beta \vdash \langle p_f, v, \eta \rangle \sim^\lambda \langle p_f, v', \eta' \rangle$ holds if the following hold:*

1. $\beta, \lfloor \kappa_r \rfloor \vdash v \sim^\lambda v'$*; and*
2. $\beta, \mathcal{T}_{p_f} \vdash \eta \sim^\lambda \eta'$*.*

3.6.2 Indistinguishability - Properties

Determining that indistinguishability of values, local variable arrays, stacks and heaps is an equivalence relation requires careful consideration on how location bijection sets are composed. Furthermore, in the presence of variable reuse (cf. high-indistinguishable variable arrays of Def. 3.31) transitivity of indistinguishability of variable arrays in fact fails unless additional conditions are imposed. These issues are discussed in this section (for proofs consult A.1.1).

In order to state transitivity of indistinguishability for values (or any component of a machine state) location bijection sets must be composed. A location bijection set β is said to be *composable* with another location set γ if for all $o \in \mathsf{Dom}(\beta)$, $\beta^{\triangleright -1}(\beta(o)) = \gamma^{\triangleleft -1}(\beta(o))$. Note that any two location sets may be assumed to be composable w.l.o.g. For composable location bijection sets we can define their composition.

Definition 3.36. *If β is composable with γ, then the location bijection set $\gamma \circ \beta$ is defined as follows (cf. Figure 3.17):*

1. $(\gamma \circ \beta)_{loc} = \gamma_{loc} \circ \beta_{loc}$[3]
2. $(\gamma \circ \beta)^{\triangleleft} = \beta^{\triangleleft}$
3. $(\gamma \circ \beta)^{\triangleright} = \gamma^{\triangleright}$

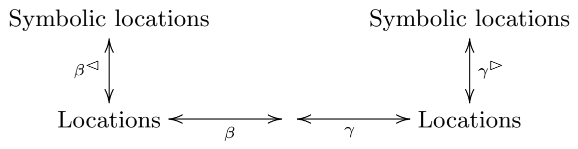


Fig. 3.17. Composition of location bijection sets

Note that, as defined, $\gamma \circ \beta$ is indeed a location bijection set. In particular, for all $o \in \mathsf{Dom}(\gamma_{loc} \circ \beta_{loc}) (\subseteq \mathsf{Dom}(\beta_{loc}))$, we have:

$$\begin{aligned}
(\gamma \circ \beta)^{\triangleleft -1}(o) &= \beta^{\triangleleft -1}(o) && \text{(Def.)} \\
&= \beta^{\triangleright -1}(\beta(o))) && (\beta \text{ loc. bij. set}) \\
&= \gamma^{\triangleleft -1}(\beta(o)) && \text{(Composability)} \\
&= \gamma^{\triangleright -1}(\gamma(\beta(o))) && (\gamma \text{ loc. bij. set}) \\
&= (\gamma \circ \beta)^{\triangleright -1}(\gamma(\beta(o)))
\end{aligned}$$

Lemma 3.37.

1. $\beta^{\mathsf{id}}, l \vdash v \sim^\lambda v$*, if $v \in \mathbb{L}$ and $l \sqsubseteq \lambda$ implies $v \in \mathsf{Dom}(\beta^{\mathsf{id}})$.*

[3] It is partial functions that are composed: $\mathsf{Dom}(\gamma_{loc} \circ \beta_{loc}) = \{o \in \mathsf{Dom}(\beta_{loc}) \mid \beta_{loc}(o) \in \mathsf{Dom}(\gamma_{loc})\}$.

2. $\beta, l \vdash v_1 \sim^\lambda v_2$ *implies* $\hat{\beta}, l \vdash v_2 \sim^\lambda v_1$.
3. *If* $\beta, l \vdash v_1 \sim^\lambda v_2$ *and* $\gamma, l \vdash v_2 \sim^\lambda v_3$, *then* $\gamma \circ \beta, l \vdash v_1 \sim^\lambda v_3$.

Proof. The first two items follow directly from a close inspection of the definition of value indistinguishability. Regarding transitivity we consider the two cases, $l \not\sqsubseteq \lambda$ and $l \sqsubseteq \lambda$. In the former case, $\gamma \circ \beta, l \vdash v_1 \sim^\lambda v_3$ holds trivially by definition of value indistinguishability. For the latter, if $v_1 = null$ or $v_1 \in \mathbb{Z}$ we resort to transitivity of equality. If $v_1 \in \mathbb{L}$, then by hypothesis and by the definition of value indistinguishability, $v_2, v_3 \in \mathbb{L}$, $\beta(v_1) = v_2$ and $\gamma(v_2) = v_3$. And by definition of $\gamma \circ \beta$, $(\gamma \circ \beta)(v_1) = \gamma(\beta(v_1)) = v_3$. Hence $\gamma \circ \beta, l \vdash v_1 \sim^\lambda v_3$. ▪

We now address local variable arrays. Variable reuse allows a public variable to be reused for storing secret information in a high security execution context. Suppose, therefore, that $l \not\sqsubseteq \lambda$, $\beta, (V_1, V_2), l \vdash \alpha_1 \sim^\lambda \alpha_2$ and $\gamma, (V_2, V_3), l \vdash \alpha_2 \sim^\lambda \alpha_3$ where, for some x, $V_1(x) = \mathtt{L}$, $V_2(x) = \mathtt{H}$ and $V_3(x) = \mathtt{L}$. Clearly it is not necessarily the case that $\gamma \circ \beta, (V_1, V_3), l \vdash \alpha_1 \sim^\lambda \alpha_3$ given that $\alpha_1(x)$ and $\alpha_3(x)$ may differ. We thus require that either V_1 or V_3 have at least the level of V_2 for this variable: $V_2(x) \sqsubseteq V_1(x)$ or $V_2(x) \sqsubseteq V_3(x)$. Of course, it remains to be seen that such a condition can be met when proving noninterference. This is indeed the case and we defer that discussion to Sec. 3.6.3. For now we state the result, namely that indistinguishability of local variable arrays is an equivalence relation (its proof is an easy consequence of lemma 3.37).

Notation 3.38

- $\mathsf{LowLoc}(\alpha, V, \lambda)$ (or simply $\mathsf{LowLoc}(\alpha)$ if the V is understood from the context) is shorthand for $\{o \mid \alpha(x) = o, V(x) \sqsubseteq \lambda\}$.
- $\mathsf{LowLoc}(\sigma, S, \lambda)$ (or simply $\mathsf{LowLoc}(\sigma)$ if the S is understood from the context) is defined as $\{o \mid \exists i \in 0..\|S\| - 1.\sigma(i) = o, S(i) \sqsubseteq \lambda\}$
- $\mathsf{LowLoc}(\eta, \beta, T, \lambda)$ (or simply $\mathsf{LowLoc}(\eta, \beta)$ if the T is understood from the context) is defined as $\{o' \mid o \in \mathsf{Dom}(\beta), \exists f \in \mathsf{Dom}(\eta(o)).(\eta(o, f) = o', T(\beta^{-1}(o), f) \sqsubseteq \lambda\}$
- $\mathsf{LowLoc}(\langle i, \alpha, \sigma, \eta \rangle, \beta, \langle i, V, S, T \rangle, \lambda)$ (or simply $\mathsf{LowLoc}(s, \beta, \lambda)$ if the $\langle i, V, S, T \rangle$ is understood from the context) is defined as $\mathsf{LowLoc}(\alpha, V, \lambda) \cup \mathsf{LowLoc}(\sigma, S, \lambda) \cup \mathsf{LowLoc}(\eta, \beta, T, \lambda)$

Lemma 3.39. *For low indistinguishability we have ($l \sqsubseteq \lambda$):*

1. $\beta^{\mathsf{id}}, V, l \vdash \alpha \sim^\lambda \alpha$, *if* $\mathsf{LowLoc}(\alpha, V, \lambda) \subseteq \mathsf{Dom}(\beta^{\mathsf{id}})$.
2. $\beta, V, l \vdash \alpha_1 \sim^\lambda \alpha_2$ *implies* $\hat{\beta}, V, l \vdash \alpha_2 \sim^\lambda \alpha_1$.
3. *If* $\beta, V, l \vdash \alpha_1 \sim^\lambda \alpha_2$, $\gamma, V, l \vdash \alpha_2 \sim^\lambda \alpha_3$, *then* $\gamma \circ \beta, V, l \vdash \alpha_1 \sim^\lambda \alpha_3$.

For high indistinguishability ($l \not\sqsubseteq \lambda$) we have:

1. $\beta^{\mathsf{id}}, (V, V), l \vdash \alpha \sim^\lambda \alpha$, *if* $\mathsf{LowLoc}(\alpha, V, \lambda) \subseteq \mathsf{Dom}(\beta^{\mathsf{id}})$.
2. $\beta, (V_1, V_2), l \vdash \alpha_1 \sim^\lambda \alpha_2$ *implies* $\hat{\beta}, (V_2, V_1), l \vdash \alpha_2 \sim^\lambda \alpha_1$.
3. *If* $\beta, (V_1, V_2), l \vdash \alpha_1 \sim^\lambda \alpha_2$, $\gamma, (V_2, V_3), l \vdash \alpha_2 \sim^\lambda \alpha_3$ *and* $\forall x \in \mathbb{X}.V_2(x) \sqsubseteq V_1(x)$ *or* $V_2(x) \sqsubseteq V_3(x)$, *then* $\gamma \circ \beta, (V_1, V_3), l \vdash \alpha_1 \sim^\lambda \alpha_3$.

The condition on β^{id} simply ensures that it is defined on the appropriate locations. This too is a condition that shall always be met given that in the proof of noninterference β^{id} is always taken to be the domain of the appropriate heap.

The case of stacks and heaps are dealt with similarly. Together these results determine that machine state indistinguishability too is an equivalence relation.

Lemma 3.40. *For low indistinguishability we have:*

1. $\beta^{\mathsf{id}} \vdash s \sim^\lambda s$, *where* $\mathsf{LowLoc}(s, \beta, \lambda) \subseteq \mathsf{Dom}(\beta^{\mathsf{id}})$ *and* $\mathsf{Ran}(\beta^{\mathsf{id}\triangleleft}), \mathsf{Ran}(\beta^{\mathsf{id}\triangleright}) \subseteq \mathsf{Dom}(\eta)$ *and* $\mathsf{Dom}(\beta^{\mathsf{id}\triangleleft}), \mathsf{Dom}(\beta^{\mathsf{id}\triangleright}) \subseteq \mathsf{Dom}(T)$.
2. $\beta \vdash s_1 \sim^\lambda s_2$ *implies* $\hat{\beta} \vdash s_2 \sim^\lambda s_1$.

3. *If $\beta \vdash s_1 \sim^\lambda s_2$, $\gamma \vdash s_2 \sim^\lambda s_3$, then $\gamma \circ \beta \vdash s_1 \sim^\lambda s_3$.*

For high indistinguishability we have:

1. $\beta^{\mathsf{id}} \vdash s \sim^\lambda s$, *where* $\mathit{LowLoc}(s, \beta, \lambda) \subseteq \mathsf{Dom}(\beta^{\mathsf{id}})$ *and* $\mathsf{Ran}(\beta^{\mathsf{id}\triangleleft}), \mathsf{Ran}(\beta^{\mathsf{id}\triangleright}) \subseteq \mathsf{Dom}(\eta)$ *and* $\mathsf{Dom}(\beta^{\mathsf{id}\triangleleft}), \mathsf{Dom}(\beta^{\mathsf{id}\triangleright}) \subseteq \mathsf{Dom}(T)$.
2. $\beta \vdash s_1 \sim^\lambda s_2$ *implies* $\hat{\beta} \vdash s_2 \sim^\lambda s_1$.
3. *Suppose* $\beta \vdash s_1 \sim^\lambda s_2$ *and* $\gamma \vdash s_2 \sim^\lambda s_3$. *Furthermore, assume* $\forall x \in \mathbb{X}. V_2(x) \sqsubseteq V_1(x)$ *or* $V_2(x) \sqsubseteq V_3(x)$. *Then* $\gamma \circ \beta \vdash s_1 \sim^\lambda s_3$.

3.6.3 Noninterference

Noninterference states that any two terminating runs of a well-typed program starting from indistinguishable initial states produce indistinguishable final states. Our notion of noninterference assumes the attacker has the ability to see the initial low variables value, and its final result (variant of termination-insensitive information flow). We do not address other covert channels (e.g. termination, timing, or abstraction-violation attacks) in this type system.

In the sequel, we assume M to be a well-typed program $\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T} \rhd ((\overrightarrow{x : \kappa}, \kappa_r), B)$. Also, we let $s_1 = \langle i, \alpha_1, \sigma_1, \eta_1 \rangle$ and $s_2 = \langle i, \alpha_2, \sigma_2, \eta_2 \rangle$ be states for M.

A program is non-interferent if two different runs that start from indistinguishable states, terminate with final indistinguishable states.

Definition 3.41 (Noninterference). *M satisfies noninterference ($\mathsf{NI}(M)$), if for every $l \sqsubseteq \lambda$ and α_1, α_2 variable arrays, v_1, v_2 values, $\eta_1, \eta_2, \eta_1', \eta_2'$ heaps and β a location bijection set:*

1. $\mathsf{mt}_M(l) = \mathcal{A}'_1, \mathcal{T}'_1, \mathcal{T}'_{p_f} \rhd ((\overrightarrow{x : \kappa}, \kappa_r, E), B)$,
2. $\langle 1, \alpha_1, \epsilon, \eta_1 \rangle \longrightarrow_B^* \langle p_f, v_1, \eta_1' \rangle$,
3. $\langle 1, \alpha_2, \epsilon, \eta_2 \rangle \longrightarrow_B^* \langle p_f, v_2, \eta_2' \rangle$,
4. $\beta, \mathcal{V}_1, l \vdash \alpha_1 \sim^\lambda \alpha_2$ *and*
5. $\beta, \mathcal{T}_1 \vdash \eta_1 \sim^\lambda \eta_2$

implies $\beta', \mathcal{T}_{p_f} \vdash \eta_1' \sim^\lambda \eta_2'$ *and* $\beta', \lfloor \kappa_r \rfloor \vdash v_1 \sim^\lambda v_2$, *for some location bijection set* $\beta' \supseteq \beta$.

With the definition of noninterference in place we now formulate the soundness result of our type system. Its proof is deferred to the last part of this section, once the appropriate supporting results are stated.

Proposition 3.42. *All well-typed methods M in JVM_C^s satisfy* $\mathsf{NI}(M)$.

The proof relies on three *unwinding lemmas* depicted in Figure 3.18((a), (b) and (d)) and whose role we now explain. The first, dubbed Equivalence in Low Level Contexts Lemma, states that related states that execute under a low security program counter proceed in lock-step and attain related states (Fig 3.18(a), where $\beta' \supseteq \beta$).

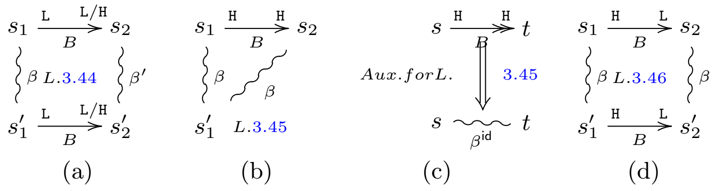


Fig. 3.18. Unwinding lemmas

$$\begin{tikzcd}
s_1 \arrow[r, twoheadrightarrow] \arrow[d, squiggly, "\beta\ L.3.44"] & s_j \arrow[r, "\beta_1^{\mathsf{id}}"'] \arrow[d, squiggly, "\beta_1"] & \cdots \arrow[r, "\beta_1^{\mathsf{id}}"'] & s_{h_1} \arrow[r] \arrow[d, squiggly, "\beta_1\ L.3.46"] & s_{h_3} \arrow[r] \arrow[d, squiggly, "\beta_1"] & \cdots \arrow[r] & s_n \arrow[d, squiggly, "\beta_m"] \\
s'_1 \arrow[r, twoheadrightarrow] & s'_j \arrow[r, "\beta_1'^{\mathsf{id}}"'] & \cdots \arrow[r, "\beta_1'^{\mathsf{id}}"'] & s'_{h_2} \arrow[r] & s'_{h_4} \arrow[r] & \cdots \arrow[r] & s'_n
\end{tikzcd}$$

Fig. 3.19. Noninterference proof scheme

Notation 3.43 if $s = \langle i, \alpha, \sigma, \eta\rangle$, then we say $\mathcal{A}_s = \mathcal{A}_i$ and $\mathtt{pc}(s) = i$.

Lemma 3.44 (One-Step Low). *Suppose*

1. $\mathcal{A}_i \sqsubseteq \lambda$;
2. $\mathtt{pc}(s_1) = \mathtt{pc}(s_2)$
3. $s_1 \longrightarrow_B s'_1$;
4. $s_2 \longrightarrow_B s'_2$; *and*
5. $\beta \vdash s_1 \sim^\lambda s_2$ *for some location bijection set* β.

Then
$\beta' \vdash s'_1 \sim^\lambda s'_2$ *and* $\mathtt{pc}(s'_1) = \mathtt{pc}(s'_2)$, *for some* $\beta' \supseteq \beta$.

Consider two runs of M, one from s_1 and another from s'_1. Computation from s_1 and s'_1 may be seen to unwind via this lemma, in lock-step, until the security context is raised to high at some states s_j and s'_j, resp. (cf. Fig 3.19). At this point, unwinding continues independently in each of the computations starting from s_j and s'_j until each of these reaches a `goto` instruction, say at state s_{h_1} for the first computation and s'_{h_2} for the second, that lowers the security level of the context back to low. Since all values manipulated in these two intermediate computations are high level values, s_j is seen to be high-indistinguishable from s_{h_1} and, likewise, s'_j is seen to be high-indistinguishable from s'_{h_2}, both facts are deduced from the following Equivalence in High Level Contexts lemma (Fig 3.18(b)).

Lemma 3.45 (One-Step High). *Let* $i_1, i_2 \in \mathsf{region}(k)$ *for some* k *such that* $\forall k' \in \mathsf{region}(k) : \mathcal{A}_{k'} \not\sqsubseteq \lambda$. *Furthermore, suppose:*

1. $\beta \vdash s_1 \sim^\lambda s'_1$ *for some location bijection set* β;
2. $s_1 \longrightarrow_B s_2$;
3. $s_2 = \langle i'_1, \alpha'_1, \sigma'_1, \eta'_1\rangle$; *and*
4. $i'_1 \in \mathsf{region}(k)$.

Then $\beta \vdash s_2 \sim^\lambda s'_1$.

The proof of lemma 3.45 relies on first proving that $\beta^{\mathsf{id}} \vdash s_1 \sim^\lambda s_2$ and then resorting to transitivity. This final step requires that we meet the condition on transitivity of variable arrays, as discussed in Sec. 3.6.2. The only problematic case would be when (1) $s_1 \longrightarrow s_2$ reuses a secret variable with public data and (2) s'_1 declares this variable to be public. Although (2) may of course hold, (1) cannot: any value written to a variable in a high security context must be secret as may be gleaned from the typing rule for `store`.

At this point both computations exit their high level regions and coincide at the unique junction point of these regions. The resulting states s_{h_3} and s'_{h_4} now become low-indistinguishable according to the following result (Fig 3.18(d)).

Lemma 3.46 (One-Step High to Low). *Let* $i_1, i_2 \in \mathsf{region}(k)$ *for some* k *such that* $\forall k' \in \mathsf{region}(k)$: $\mathcal{A}_{k'} \not\sqsubseteq \lambda$. *Suppose:*

1. $s_1 \longrightarrow_B s'_1$ *and* $s_2 \longrightarrow_B s'_2$;

2. $\beta \vdash s_1 \sim^\lambda s_2$, for some location bijection set β; and
3. $s_1' = \langle i_1', \alpha_1', \sigma_1', \eta_1' \rangle$ and $s_2' = \langle i_2', \alpha_2', \sigma_2', \eta_2' \rangle$;
4. and $\mathcal{A}_{i_1'} = \mathcal{A}_{i_2'}$ and $\mathcal{A}_{i_1'} \sqsubseteq \lambda$.

Then $i_1' = \mathsf{jun}(k) = i_2'$ and $\beta \vdash s_1' \sim^\lambda s_2'$.

Noninterference Proof

The proof of noninterference (Proposition 3.42) follows.

Proof. Consider the two runs of M depicted in Figure 3.19 where

- $s_1 = \langle 1, \alpha_1, \sigma_1, \eta_1 \rangle$ and $s_1' = \langle 1, \alpha_1', \sigma_1', \eta_1' \rangle$ with $\sigma_1 = \sigma_1' = \epsilon$.
- $s_n = \langle p_f, v, \eta_n \rangle$ and $s_n' = \langle p_f, v', \eta_m' \rangle$.
- $\beta \vdash s_1 \sim^\lambda s_1'$ for some location bijection set β .

First some notation: we write $\mathsf{region}(k) = \mathtt{H}$ if $\forall j \in \mathsf{region}(k).\mathcal{A}_j \not\sqsubseteq \lambda$.

Now to the proof. Starting from s_1 and s_1', repeatedly apply Case 1 and Case 2, described below, until the final states are reached.

1. **Case 1.** If the current instruction level is $\sqsubseteq \lambda$, repeatedly apply Lemma 3.44 until it is no longer possible. Suppose s_j and s_j' are the states reached.
   We have two cases to consider:
   - Suppose $s_j = s_n = \langle p_f, v, \eta_n \rangle$ and $s_j' = s_m' = \langle p_f, v', \eta_m' \rangle$. Then this states are final states and by Lemma 3.44 we have $\beta', \lfloor \kappa_r \rfloor \vdash v \sim^\lambda v'$ and $\beta', \mathcal{T}_{p_f} \vdash \eta_n \sim^\lambda \eta_m'$ for some location bijection set $\beta' \supseteq \beta$.
   - Otherwise apply Case 2. By Lemma 3.44 we know that $\mathtt{pc}(s_j) = \mathtt{pc}(s_j')$ and by previous case s_j and s_j' are no final states.
2. **Case 2.** If the current instruction level is $\not\sqsubseteq \lambda$. Let us call s_j and s_j' the current states, we know that $\mathcal{A}_{s_j}, \mathcal{A}_{s_j'} \not\sqsubseteq \lambda$ and $\beta_1 \vdash s_j \sim^\lambda s_j'$ for some $\beta_1 \supseteq \beta$. As a consequence, there exists a program point k in $\mathsf{Dom}(B)^\sharp$ such that k was the value of the program counter for s_{j-1} and s_{j-1}'. Furthermore, $k \mapsto g$ and $k \mapsto g'$, for $g = \mathtt{pc}(s_{j-1})$ and $g' = \mathtt{pc}(s_{j-1}')$, by the definition of the Successor Relation. By SOAP property (3.10(i)), both $g, g' \in \mathsf{region}(k)$. Furthermore, $\mathsf{region}(k) = \mathtt{H}$ follows from the conditions of the typing rule T-If, T-GtFld, T-PtFld or T-InvkVrtl. Therefore the hypothesis of Lemma 3.45 is satisfied.
   Now, we repeatedly apply Lemma 3.45 to the first program run in Figure 3.19 until it is no longer possible. Let us call s_{h_1} the reached state. Then by (the repeated application of) Lemma 3.45, $\beta_1 \vdash s_{h_1} \sim^\lambda s_j'$. By symmetry, $\hat{\beta_1} \vdash s_j' \sim^\lambda s_{h_1}$. Applying a similar reasoning to the program run at the bottom of Figure 3.19 yields the existence of a state s_{h_2}' such that $\hat{\beta_1} \vdash s_{h_2}' \sim^\lambda s_{h_1}$. Finally, by resorting to symmetry once more we conclude $\hat{\hat{\beta_1}} \vdash s_{h_1} \sim^\lambda s_{h_2}'$. Then $\beta_1 \vdash s_{h_1} \sim^\lambda s_{h_2}'$.
   We have two cases to consider:
   - Suppose $s_{h_1} = s_n = \langle p_f, v, \eta_n \rangle$ and $s_{h_2}' = s_m' = \langle p_f, v', \eta_m' \rangle$ are the states reached. Then this states are final states. By the above reasoning, we know that the predecessors states of s_{h_1} and s_{h_2}' are indistinguishable, called s_{h_3} and s_{h_4}' . The heap and the stack are indistinguishable. The instructions that generate final states no modify the heap then we can affirm $\beta_1, \mathcal{T}_{p_f} \vdash \eta_n \sim^\lambda \eta_m'$. Furthermore, v and v' are the stack top in the predecessors states, then $\beta_1, \lfloor \mathcal{S}_{h_3}(0) \sqcup \mathcal{S}_{h_4}'(0) \rfloor \vdash v \sim^\lambda v'$, and by typed schemes $\mathcal{S}_{h_3}(0), \mathcal{S}_{h_4}'(0) \sqsubseteq \kappa_r$. Then we have $\beta_1, \lfloor \kappa_r \rfloor \vdash v \sim^\lambda v'$.
   - In other case, by Lemma 3.45 we know that $\mathcal{A}_{s_{h_1}}, \mathcal{A}_{s_{h_2}'} \not\sqsubseteq \lambda$. Furthermore, for all h_3 and h_4 such that $h_1 \mapsto h_3$ and $h_2 \mapsto h_4$, we have $\mathcal{A}_{h_3}, \mathcal{A}_{h_4} \sqsubseteq \lambda$. Hence there exist h_3 and h_4 such that $h_1 \mapsto h_3$ and $h_2 \mapsto h_4$. Finally, note that by Lemma 3.45, $\mathtt{pc}(s_{h_1}), \mathtt{pc}(s_{h_2}') \in \mathsf{region}(k) = \mathtt{H}$. Therefore, we are in condition of applying Lemma 3.46 to deduce $\beta_1 \vdash s_{h_3} \sim^\lambda s_{h_4}'$.

We repeat the argument, namely applying Lemma 3.44, Case 1 and Case 2 until it is no longer possible. Given that the derivations are finite, eventually final states are reached.

▪

Remark 3.47. Noninterference may be proved for any program which is well-typed rather than just well-typed methods: the requirement that execution begin at a state where the stack is empty is not necessary for the result to hold.

3.7 Discussion

We have presented a type system for ensuring secure information flow in a JVM-like language that allows instances of a class to have fields with security levels depending on the context in which they were instantiated. This differs over the extant approach of assigning a global fixed security level to a field, thus improving the precision of the analysis as described in the motivation 3.1.

We want to emphasize that the type system was developed incrementally: we went from JVM^s to JVM^s_E and then finally JVM^s_C. We proved noninterference for each of the bytecode subsets. The proof of noninterference was developed incrementally, too. Moreover, the infrastructure needed for prove noninterference (definition of indistinguishability and unwinding lemmas) was reused in each step of development of the type system.

3.7.1 Extensions

Support for threads seems to be absent in the literature on IFA for bytecode languages (a recent exception being [27]) and would be welcome. The same applies to declassification. Regarding the latter, it should be mentioned that some characteristics of low level code such as variable reuse enhance the capabilities of an attacker to declassify data.

Our current efforts are geared toward enriching the core bytecode language while maintaining our results. We can treat arrays in the same way as `Jif` does. That is, like `Java`, `Jif` treats arrays as normal objects. Pointers to arrays can be assigned, dereferenced or cascaded to form multi-dimensional arrays. The type system handles regular array read and update operations. Out-of-bound accesses will cause an `OutOfBoundException`. This exception will leak information about the array size and the index being used. So they must be caught and handled properly.

In Section 2.2 we mentioned that bytecode instruction set is factorisable into 12 instruction mnemonics. A single instruction mnemonic is not considered in JVM^s_C: the type conversion (cast) and type checking instructions. We can treat this class of instructions as skip instructions (instruction does not have any effect on the state of the program). Conversion and type checking instructions do not have any effect in the information flow analysis.

To sum up, although less significant, these are worthy contributions towards minimizing the assumptions on the code that is to be analyzed hence furthering its span of applicability. In order to achieve an acceptable degree of usability, the information flow type system could be based on preliminary static analyses that provide a more accurate approximation of the control flow graph of the program. Typically, the preliminary analyses will perform safety analyses such as class analysis, null pointer analysis, exception analysis and escape analysis. For example, null pointer analysis could use to indicate whether the execution of a `putfield` instruction will be normal or there will be an exception. This analysis drastically improve the quality of the approximation of the control dependence regions.

3.7.2 Related work

This section presents work previously done on information flow analysis. First present an overview of the early work leading to information flow analysis. Second, we review some of the relevant literature (mostly) on type-based IFA for low-level languages and Java. A survey of research on information flow, highlighting all the open areas of research, is given by Sabelfeld and Hedin [98].

Information Flow

Denning and Denning [53, 54] point out that information flow can be tracked conservatively using static program analysis. A security lattice model is used to classify data sensitivity. Data flow is allowed only when labels on the data units along the flow path increase monotonically. This way information of a certain security level is kept within the boundary given by the labels in the program. The system is formulated in a way that programs can be checked and certified mechanically. Based on Denning and Denning's work, Volpano and Smith first analyze secure information flow using a security type system [116, 117]. They consider a sequential imperative language with procedures. Variables, method parameters and method returns are given level constants indicating their sensitivity. A typing rule is given for each language construct to ensure that there is no information flow from high security input to low security output according to the labels on the data. The analysis is conservative in that a well-typed program is guaranteed not to leak sensitive data. The type system is proved to ensure non-interference. An inference system is designed to infer security labels automatically. The language in this work is rather simple. It does not support object-oriented features. No polymorphism on procedures is provided.

Following Cohen's notion of strong dependency [45], Goguen and Meseguer [62] introduced the general concept of non-interference to formalize security policies. A formal model for checking security properties with regard to given confidentiality policies is given using simulation theory. Non-interference is used to define security guarantees by a lot of work. In this work, our type system is proved to preserve non-interference.

Myers presents Jif [82] a rich subset of Java with type annotations together with a type system ensuring secure information flow based on the decentralized label model [84]. Some advanced features of Jif include label polymorphism, run-time label checking and automatic label interference. Soundness of Jif is not proved.

Noninterference properties are not safety properties. They are defined as properties of pairs of computations rather than individual ones. Standard automatic software model checking tools cannot reason about multiple runs of a program. They deal exclusively with safety properties which involves reasoning about a single run [48, 111]. Barthe, D'Argenio and Rezk [23] explore extensively self composition, a technique that allows to reduce information flow policies to properties of single executions. Self composition provides methods to establish security properties for languages for which no information flow type system is known. In [94] are proposed specifications for different information flow policies using self composition in several frameworks, both for sequential and concurrent non-deterministic programming languages. Characterization of security are given in Hoare logic (termination-insensitive security), weakest precondition calculus (termination-sensitive security), separation logic (pointer programs), temporal logics (CTL and LTL) and remarks on how to combine verification method using self composition with type systems.

Hunt and Sands [65] present a flow-sensitive type system for tracking information flow in a simple While language which allow assign to variables different security types at different points in the programs. This type system preserves non-interference. Finally they show, for any program typeable in one of these systems, how to construct an equivalent program which is typeable in a simple flow-insensitive system. Extend this work with robust declassification involves similar restricts in the attacks (w.r.t. variable reuse) that the present in this work.

Information Flow for Low-Level Languages

Kobayashi and Shirane [68] develop an IFA type system for a simple subset of JVM bytecode. Their system verifies a noninterference property and encompasses information leaking based on timing channels. Object creation and exceptions are not considered.

Lanet et al. [35] develop a method to detect illicit flows for a sequential fragment of the JVM. In a nutshell, they proceed by specifying in the SMV model checker [77] a symbolic transition semantics of the JVM that manipulates security levels, and by verifying that an invariant that captures the absence of illicit flows is maintained throughout the abstract program execution. Their analysis is more flexible than ours, in that it accepts programs such as $y_{\mathtt{L}} = x_{\mathtt{H}}; y_{\mathtt{L}} = 0$. However, they do not provide a proof of non-interference.

The approach of Lanet et al. has been refined by Bernardeschi and De Francesco [31] for a subset of the JVM that includes jumps, subroutines but no exceptions.

Barthe, Basu and Rezk [21] define a simple JVM bytecode type system for IFA. They prove preservation of noninterference for a compilation function from a core imperative language to bytecode. Barthe and Rezk [25] extend the type system to handle objects and method calls. Later, Barthe, Naumann and Rezk [26] present a type-preserving compilation for a Java-like language with objects and exceptions. In this work, they study the formal connection between security policies on the source code and properties of the compiled code using certified compilers. They extend the type system of [17] to include exceptions, and prove that if the source program is typable, the compiled bytecode is also typable. A non-interference result is obtained by applying the analysis to connect the source language to the low-level byte code in Barthe and Rezk' work [21], which is proved to enforce non-interference. Further details on this work may be found in the thesis of T. Rezk [94]. All these works assume fields have a fixed security level. Less importantly, they do not allow variable reuse and have a less precise management of information flow of the operand stack. Barthe, Rezk, Russo and Sabelfeld [27] considers a modular method to devise sound enforcement mechanisms for multi-threaded programs. The central idea of these works is to constrain the behavior of the scheduler so that it does not leak information; it is achieved by giving to the scheduler access to the security levels of program points, and by requiring that the choice of the thread to be executed respects appropriate conditions. The type system for the concurrent language is defined in a modular fashion from the type system for the sequential language, and the soundness of the concurrent type system is derived from unwinding lemmas for the sequential type system.

The method presented by Bernardeschi et al. [32] relies on the type-level abstract interpretation of standard bytecode verification to detect illegal information flows. Verification is performed on the code output by a transformation algorithm applied to the original code. If verification succeeds, then the original code is shown to be secure. The main advantage of the proposed method is that it does not rely on any extra verification tool. Moreover, verification is performed inside the Java environment. The main disadvantage of their approach is that the analysis is monomorphic. It should also be mentioned that all fields of all instances of a class are assumed to have the same, fixed security level.

Also, there is work on IFA of Typed Assembly Language (TAL) [38, 124] although these languages do not provide primitives for object creation nor exceptions. They do however deal with register reuse [124] and polymorphic assembly code [38, 124]. Yu and Islam [124] develop a type system for a rich assembly language (similar to SIFTAL) and prove type-preserving compilation for an imperative language with procedures. But, the stacks not can be manipulated in the high contexts. Bonelli, Compagnoni and Medel [38] development a sound information flow type system for a simple assembly language called SIFTAL. SIFTAL has primitives to raise and lower the security level of regions for enforce structured control flow. They do not assume anything about the use of the stack by the programs, and verifies that not leaking of information is produced by stack manipulation. Also, they not assume the existence of trusted functions that obtain the code regions. Furthermore, SIFTAL has not reuse of registers. In a recent work, Bonelli and Molinari [40] present a type-preserving compilation of a simple high level programming language to SIFTAL.

Genaim and Spoto [57] present a flow and context sensitive compositional information flow analysis for full (mono-threaded) Java bytecode (including methods exceptions and dynamic heap allocation). The analysis is based on the transformation of bytecode into a control-flow graph of basic code blocks and use boolean functions to represent sets of information flows for which binary decision diagrams become applicable.

Our notation is based on work by Kobayashi and Shirane [68] and [56]. The former develops an IFA type system for a simple subset of JVM bytecode and encompasses information leaking based on timing channels. The second studies correctness of JVM's bytecode verifier.

Ghindici [58] present a sound technique for enforcing information flow model in open systems. It relies on static analysis, abstract interpretation [47] and points-to analysis [122]. To adapt to the dynamic downloading from untrusted sources and through insecure channels, certify JVM bytecode, and the verification is performed at loading time. Due to the limited resources of open systems, they split the verification process in two steps: first, an external analysis, that could be performed on any computer and which computes, for each method, a flow signature containing all possible flows within the method, and proof elements which are shipped with the code; second, an embedded analysis, which certifies the flow signature, using the proof, at loading time. The flow signature is computed using abstract interpretation and type inference techniques. Hence, they have designed for the external analysis an automatic type inference system for object- oriented languages. It system does both inter-procedural and intra-procedural inference, in contrast with previous work on information flow inference for object-oriented languages which supports only intra-procedural inference. Moreover, they perform modular inference, e.g., each flow signature is computed only once. Furthermore, Ghindici have implemented two tools: the prover STAN and the verifier VSTAN. STAN implements the information flow model; it is a abstract analyser that allows static alias analysis and information flow certification. STAN statically computes the type inference and annotates the .class files with proof elements and flow signatures verifiable at loading time. VSTAN check information flow at loading time. VSTAN verifies JVM bytecode annotated by STAN [58].

4

Robust Declassification for Java Bytecode

The starting point of this chapter is the core fragment of bytecode JVM_C^s extended with a declassification instruction `declassify` l, where l is a security level (see Section 4.3). The effect of this instruction is to declassify the top element of the operand stack of the JVM run-time system.

We study a notion of robust declassification [85] for JVM_C^s and discuss several subtle ways in which the standard notion of robust declassification fails. Particularly, this fails in the presence of flow-sensitive types and low-level stack manipulation primitives. Under an attacker that observes and manipulates data, we address the requirements that programs must meet in order for robustness to be enjoyed. We propose a type system for enforcing robust declassification in JVM_C^s and prove that this system is sound.

Structure. Section 4.1 introduces motivating examples. It is followed by the attacker model. The type system is presented in Section 4.3. Section 4.4 addresses noninterference and robust declassification.

4.1 Motivating Examples

Information flow policies in terms of noninterference ensure the absence of information channels from private to public data. Although an intuitively appealing information security policy, it has been recognized to be too strict for practical applications since many of them do allow some form of information declassification or downgrading. Examples are: releasing the average salary from a secret database of salaries for statistical purposes, a password checking program, releasing an encrypted secret, etc. This calls for relaxed notions of noninterference where primitives for information declassification are adopted. At the same time we are also interested in mobile code. Currently a large amount of code is distributed over the Internet largely in the form of bytecode [75] or similar low-level languages. Direct verification of security properties of bytecode allows the code consumer to be assured that its security policy is upheld. It should also be stressed that most efforts on information flow target high-level languages yet compilation can produce a program which does not necessarily satisfy the same security properties. Analysis at the bytecode level has the benefit of independence of the mechanism (compiled, handwritten, code generated) that generated it.

An important issue that arises in mobile code (or any code) that provides a declassification mechanism is whether it can be exploited affecting the data that is declassified or whether declassification happens at all. If this does not arise we declare our declassification mechanism *robust*. We briefly illustrate that with an example taken from [85] and translated to bytecode.

1. `load` $x_{\mathtt{L}}$
2. `if` 6
3. `load` $z_{\mathtt{H}}$
4. `declassify L`
5. `store` $y_{\mathtt{L}}$
6. $\cdots$

This program releases private information held in $z_{\mathtt{H}}$ to a publicly readable variable $y_{\mathtt{L}}$ if the variable $x_{\mathtt{L}}$ is true, and otherwise has no effect. Here, `declassify` explicitly indicates a release from secret to public of the level of the topmost element on the stack. As a consequence, this program does not enjoy noninterference. However, we can observe something more on the reliability of the declassification mechanism. Indeed, an attacker can affect whether information is declassified at all (assuming she can change the value of $x_{\mathtt{L}}$) or can affect the information that is declassified (assuming she can change the value of $z_{\mathtt{H}}$). This program is therefore declared not to be robust under any of these two assumptions.

We consider that attackers may change system behavior by injecting new code into the program. Such attacker-controlled low-integrity computation, called **active attacker**, may be interspersed with trusted high-integrity code. To distinguish the two, the high-integrity code is represented as a program in which some statements are missing, replaced by holes and denoted with bullets $\bullet$. The idea is that the holes are places where the attacker can insert arbitrary low-integrity code. There may be multiple holes (possibly none) in the high-integrity code, represented by the notation $\overrightarrow{\bullet}$. Such expressions are called *high-integrity contexts*, written $B[\overrightarrow{\bullet}]$, and formally defined as programs in which the set of bytecode instructions is extended with a distinguished symbol "$\bullet$". These holes can be replaced by a vector of attacker fragments $\overrightarrow{a}$ to obtain a complete method $B[\overrightarrow{a}]$. An attacker is thus a vector of such code fragments. An active attacker is any attack vector that fills some hole in a way that changes the original program behavior.

One way an attacker can cause information leak is to directly violate confidentiality and integrity (eg. using declassify in its own code). However, one is interested in how an attacker can exploit insecure information flow in the trusted code. Therefore, we restrict our attention to attacks that are *active* in this sense (cf. Def. 4.3). We discuss some examples[1].

Example 4.1. Examples of user code subject to attacks.

```
1 load x_H
2 load y_H
3 [•]
4 declassify L
5 ···
```
(a)

```
1 [•]
2 load x_H
3 declassify L
4 ···
```
(b)

```
1 [•]
2 load x_H
3 declassify L
4 return
```
(c)

```
1 [•]
2 load  x_H
3 if  5
4 declassify L
5 ···
```
(d)

In (a), assuming $x_{\mathtt{H}}$ and $y_{\mathtt{H}}$ are confidential, at line 4 the value of $y_{\mathtt{H}}$ is declassified. If the attacker consists of a `pop` instruction, then declassification of the value of $x_{\mathtt{H}}$ is forced. Likewise the attacker could push an element on the stack rather than remove it. In this case it is the newly pushed element that is declassified. Let us consider another example.

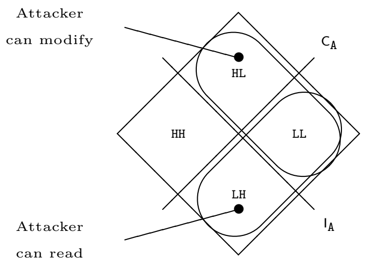
In (b) the security level of $x_{\mathtt{H}}$ at 2 is assumed to be confidential. If the attacker inserts `load` $y_{\mathtt{H}}$; `goto` 3 where $y_{\mathtt{H}}$ holds some high confidentiality data, then by affecting the control-flow $y_{\mathtt{H}}$ is now declassified.

If the code is that of (c), then by inserting the code `load` y; `return` declassification of $x_{\mathtt{H}}$ is avoided.

One final consideration is that we must also restrict variable reuse in low-integrity contexts. Consider the code excerpt (d) where x is assumed high-integrity at 2 and y low-integrity at 2. Line 4 attempts to declassify depending on the value of x (a high-integrity value). Suppose the attacker reuses x with a low-integrity value, such as in `load` y; `store` x. Then in line 4 a declassification is attempted depending on the value of the variable y (a low-integrity value). Therefore the attacker can manipulate the condition in line 3 which decides whether declassification is performed or not. This situation is avoided by restricting attackers from manipulating high-integrity variables such as x above.

Summing up, our definition of active attack requires that attacks not include `return` nor `declassify` instructions. Likewise, we require all jumps to be local to the attackers code. The size of the stack before the attacker code is executed should be the same after it is executed. Finally, high-integrity variables and stack elements should not be modifiable. These observations are made precise in Def. 4.3 and incorporated into the typing schemes for our bytecode language (Sec. 4.3).

[1] In these examples, for expository purposes, the argument of `declassify` is `L`. However these labels are actually pairs of levels (cf. Sec. 4.2).



$$\begin{aligned} \mathtt{H}_C &= \{\kappa \mid \mathsf{C}_\kappa \not\sqsubseteq_C \mathsf{C}_{\mathtt{A}}\} \\ \mathtt{L}_C &= \{\kappa \mid \mathsf{C}_\kappa \sqsubseteq_C \mathsf{C}_{\mathtt{A}}\} \\[1em] \mathtt{H}_I &= \{\kappa \mid \mathsf{I}_{\mathtt{A}} \not\sqsubseteq_I \mathsf{I}_\kappa\} \\ \mathtt{L}_I &= \{\kappa \mid \mathsf{I}_{\mathtt{A}} \sqsubseteq_I \mathsf{I}_\kappa\} \end{aligned}$$

Fig. 4.1. Attacker's view of lattice $\mathcal{L}$.

4.2 Attacker Model

Since robustness of a system depends on the power an attacker has to affect the execution of the system, we follow [85] and introduce *integrity* labels in our model. High-integrity information and code are trusted and assumed not to be affected by the attacker while low-integrity information and code are untrusted and assumed to be under the control of the attacker. Both confidentiality and integrity labels are merged into a single lattice of *security levels*.

We assume given a lattice of confidentiality levels $\mathcal{L}_C$ with ordering $\sqsubseteq_C$ and a lattice of integrity levels $\mathcal{L}_I$ with ordering $\sqsubseteq_I$. We use $\bot_C$ and $\top_C$ for the bottom and top elements of $\mathcal{L}_C$ and similarly for those of $\mathcal{L}_I$. We write l_C for elements of the former and l_I for elements of the latter and $\sqcup_C$ and $\sqcup_I$, resp., for the induced join. In this work, we assume the lattice to be $\{\mathtt{L}, \mathtt{H}\}$; $\mathtt{L} \sqsubseteq_C \mathtt{H}$; and $\mathtt{H} \sqsubseteq_I \mathtt{L}$.

Now, a **security level** is a pair (l_C, l_I).

And, a **security label** is an expression of the form

$$\langle \{a_1, \ldots, a_n\}, (l_C, l_I)\rangle$$

where we recall from Sec. 3.3 that a_i, $i \in 0..n$, ranges over the set of symbolic locations *SymLoc*. If $n = 0$, then the security label is $\langle \emptyset, (l_C, l_I)\rangle$. We say a_i (for each $i \in 1..n$) and (l_C, l_I) are the symbolic locations and level of the security label, respectively. We occasionally write $\langle \_, (l_C, l_I)\rangle$, or (l_C, l_I), when the set of symbolic locations is irrelevant. Also, we occasionally write l instead of (l_C, l_I).

The security labels are elements of a lattice $\mathcal{L}$ and are typically denoted using letters κ, κ_i. The partial order $\sqsubseteq$ on $\mathcal{L}$ is defined as:

$$\langle R_1, (l_C, l_I)\rangle \sqsubseteq \langle R_2, (l'_C, l'_I)\rangle \text{ iff } R_1 \subseteq R_2,\ l_C \sqsubseteq_C l'_C \text{ and } l_I \sqsubseteq_I l'_I.$$

The join on security labels is:

$$\langle R_1, (l_C, l_I)\rangle \sqcup \langle R_2, (l'_C, l'_I)\rangle = \langle R_1 \cup R_2, (l_C \sqcup_C l'_C, l_I \sqcup_I l'_I)\rangle.$$

The projections of the first and second component of the level of a security label are:

$$\mathsf{C}(\langle R_1, (l_C, l_I)\rangle) = l_C \text{ and } \mathsf{I}(\langle R_1, (l_C, l_I)\rangle) = l_I.$$

Also, we occasionally write C_κ for $\mathsf{C}(\kappa)$ and I_κ for $\mathsf{I}(\kappa)$.

The *power* of an attacker is described by a security level $\mathtt{A}$, where $\mathsf{C}_{\mathtt{A}}$ is the confidentiality level of data the attacker is expected to be able to read, and $\mathsf{I}_{\mathtt{A}}$ is the integrity level of data or code that the attacker is expected to be able to affect. Thus, the robustness of a system is w.r.t. the attacker security level $\mathtt{A}$.

The attacker security level $\mathtt{A}$ determines both high and low-confidentiality ($\mathtt{H}_C$ and $\mathtt{L}_C$) areas and high and low-integrity areas ($\mathtt{H}_I$ and $\mathtt{L}_I$) in the security lattice $\mathcal{L}$, as depicted in Fig. 4.1. We can identify four key areas of the lattice:

1. $\mathtt{LL} \simeq \mathtt{L}_C \cap \mathtt{L}_I$. The $\mathtt{LL}$ region is under complete control of the attacker: she can read and modify data in this region.
2. $\mathtt{LH} \simeq \mathtt{L}_C \cap \mathtt{H}_I$. Data in the $\mathtt{LH}$ region may be read but not modified by the attacker.

3. $\mathtt{HH} \simeq \mathtt{H}_C \cap \mathtt{H}_I$. The attacker may not read nor modify data in the region HH.
4. $\mathtt{HL} \simeq \mathtt{H}_C \cap \mathtt{L}_I$. Data in the HL region may not be inspected but can be modified by the attacker.

4.3 The Bytecode Language and its Type System

Our language is JVM^s_C but with the addition of the following instruction:

$$\texttt{declassify}\ l$$

which *declassifies* (or downgrades) to l the security level of value on the top of the operand stack. This instruction does not have any run-time effect on the state of the program.

Definition 4.2 (Method). *A method $M[\overrightarrow{\bullet}]$ is a pair $((\overrightarrow{x : \kappa}, \kappa_r, E), B[\overrightarrow{\bullet}])$, where $(\overrightarrow{x : \kappa}, \kappa_r, E)$ is a method type and $B[\overrightarrow{\bullet}]$ is the method's body.*

4.3.1 Typing Schemes

Methods $M[\overrightarrow{\bullet}]$ are typed by means of a **typing judgement**: $\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T} \rhd M[\overrightarrow{\bullet}]$. Recall from Def. 3.7 that the tuple $(\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T})$ is a typing context. Henceforth we turn to the definitions of well-typing method (Definition 3.23).

$\mathcal{A}$ indicates the level of each instruction. If $\mathcal{A}_i$ is high at some program point because $\mathsf{C}(\mathcal{A}_i) \in \mathtt{H}_C$, then an attacker might learn secrets from the fact that execution reached that point. If $\mathcal{A}_i$ is high because $\mathsf{I}(\mathcal{A}_i) \in \mathtt{L}_I$, the attacker might be able to affect whether control reaches that point.

Just as in JVM^s_C we adopt three assumptions that, without sacrificing expressiveness, simplify our analysis:

- The bytecode passes the bytecode verifier.
- Method body $B[\overrightarrow{\bullet}]$ has a unique `return` instruction.
- Predecessors of a junction point are always `goto`, `return` or err_i instructions.

Additionally, the typing schemes rely on region and jun satisfying the SOAP properties, as discussed for JVM^s_C.

The typing schemes are given in Fig. 4.2, Fig. 4.3, and Fig. 4.4 and are described below.

We now describe the typing schemes. Except for some additional conditions to be discussed below, these rules are identical to those of JVM^s_C. The new conditions avoid high integrity variable reuse (T-Str, condition 6) and popping high integrity values off the stack (for example, condition 6 of T-Str and T-Pop and condition 5 of T-PrimOP) in low integrity contexts. These constraints are necessary for proving that the type system guarantees robustness.

T-Declassify states that only high-integrity data is allowed to be declassified and that declassification might only occur at a high-integrity security environment. Because the possible flow origins is within the high-integrity area of the lattice, the attacker (who can only affect the low-integrity area of the lattice) cannot influence uses of the declassification mechanism and therefore cannot exploit it.

Regarding T-Hole, for robustness, it is important that holes not be placed at program points where the type environment is high-confidentiality, because then the attacker would be able to directly observe implicit flows to the hole and could therefore cause information to leak even without a `declassify`.

(T-Declassify)
$$\dfrac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{declassify}\ l \\ (\mathcal{S}_i \backslash 0) \leq_{\mathcal{A}_i} (\mathcal{S}_{i+1} \backslash 0) \\ \mathcal{A}_i \sqcup l \sqsubseteq \mathcal{S}_{i+1}(0) \\ \mathsf{l}(\mathcal{S}_i(0)) = \mathsf{l}(l) \\ \mathsf{l}(\mathcal{A}_i), \mathsf{l}(\mathcal{S}_i(0)) \in \mathtt{H}_I \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

(T-Hole)
$$\dfrac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = [\bullet] \\ \mathcal{A}_i \in \mathtt{L}_C \\ \mathcal{S}_i \leq_{(\perp_C, \mathsf{l}(\mathtt{A}))} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

(T-PrimOp)
$$\dfrac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{prim}\ op \\ \kappa' \sqcup \kappa'' \sqsubseteq \mathcal{S}_{i+1}(0) \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \kappa' \cdot \kappa'' \cdot (\mathcal{S}_{i+1} \backslash 0) \\ \mathcal{A}_i \sqsubseteq \kappa', \kappa'' \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \\ \mathcal{S}_i(0), \mathcal{S}_i(1) \in \mathtt{L}_I,\ \mathit{if}\ \mathcal{A}_i \in \mathtt{L}_I \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

(T-If)
$$\dfrac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{if}\ i' \\ \mathcal{V}_i \leq \mathcal{V}_{i+1}, \mathcal{V}_{i'} \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \kappa \cdot \mathcal{S}_{i+1}, \kappa \cdot \mathcal{S}_{i'} \\ \mathcal{A}_i \sqsubseteq \kappa \\ \mathcal{T}_i \leq \mathcal{T}_{i+1}, \mathcal{T}_{i'} \\ \forall k \in \mathtt{region}(i).\kappa \sqsubseteq \mathcal{A}_k \\ \mathcal{S}_i(0) \in \mathtt{L}_I,\ \mathit{if}\ \mathcal{A}_i \in \mathtt{L}_I \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

(T-Ret)
$$\dfrac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{return} \\ \mathcal{S}_i(0) \sqcup \mathcal{A}_i \sqsubseteq \kappa_r \\ \mathcal{T}_i \leq \mathcal{T}_{p_f} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

(T-Goto)
$$\dfrac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{goto}\ i' \\ \mathcal{V}_i \leq \mathcal{V}_{i'} \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\ \mathcal{T}_i \leq \mathcal{T}_{i'} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

(T-Pop)
$$\dfrac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{pop} \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \kappa \cdot \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \\ \mathcal{A}_i \sqsubseteq \kappa \\ \mathcal{S}_i(0) \in \mathtt{L}_I,\ \mathit{if}\ \mathcal{A}_i \in \mathtt{L}_I \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

(T-Erri)
$$\dfrac{\begin{array}{l} B(err_i) \\ \mathcal{S}_i(0) \sqcup \mathcal{A}_i \sqsubseteq \kappa_r \\ \mathcal{T}_i \leq \mathcal{T}_{p_f} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

(T-Load)
$$\dfrac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{load}\ x \\ \mathcal{A}_i \sqcup \mathcal{V}_i(x) \cdot \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

(T-Str)
$$\dfrac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{store}\ x \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{V}_{i+1}(x) \cdot \mathcal{S}_{i+1} \\ \mathcal{V}_i \backslash x \leq \mathcal{V}_{i+1} \backslash x \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \\ \mathcal{A}_i \sqsubseteq \mathcal{V}_{i+1}(x) \\ \mathcal{S}_i(0), \mathcal{V}_i(x) \in \mathtt{L}_I,\ \mathit{if}\ \mathcal{A}_i \in \mathtt{L}_I \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

(T-Push)
$$\dfrac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{push}\ n \\ \mathcal{A}_i \cdot \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

(T-New)
$$\dfrac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{new}\ C \\ a = \mathsf{Fresh}(\mathcal{T}_i) \\ \mathcal{T}_i \oplus \{a \mapsto [f_1 : \mathtt{ft}_{\mathcal{A}_i}(f_1), \ldots, f_n : \mathtt{ft}_{\mathcal{A}_i}(f_n)]\} \leq \mathcal{T}_{i+1} \\ \langle \{a\}, \mathcal{A}_i \rangle \cdot \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

Fig. 4.2. Full set of typing schemes I

$$\text{(T-ThrH)}\quad \frac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{throw} \\ \mathsf{Handler}(i) = i' \\ \mathcal{S}_i(0) \sqsubseteq \mathcal{A}_{i'} \\ \mathcal{S}_i(0) \cdot \epsilon \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\ \mathcal{V}_i \leq \mathcal{V}_{i'} \\ \mathcal{T}_i \leq \mathcal{T}_{i'} \\ \forall j \in \mathsf{Dom}(\mathcal{S}_i).\mathcal{A}_i \sqsubseteq \mathcal{S}_i(j) \\ \forall j \in \mathsf{Dom}(\mathcal{S}_i).\mathcal{S}_i(j) \in \mathtt{L}_I,\ \mathit{if}\ \mathcal{A}_i \in \mathtt{L}_I \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

$$\text{(T-ThrNoH)}\quad \frac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{throw} \\ \mathsf{Handler}(i) \uparrow \\ \mathcal{A}_i \sqcup \mathcal{S}_i(0) \sqsubseteq \kappa_r \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{err_i} \\ \mathcal{V}_i \leq \mathcal{V}_{err_i} \\ \mathcal{T}_i \leq \mathcal{T}_{err_i} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

$$\text{(T-PtFld)}\quad \frac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{putfield}\ f \\ \mathsf{Handler}(i) \downarrow \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \langle R_1, l_1 \rangle \cdot \langle R, l \rangle \cdot \mathcal{S}_{i+1} \\ l_1 \sqcup l \sqsubseteq \bigsqcap_{a \in R} \mathcal{T}_i(a, f) \\ \mathit{for\ all}\ a \in R.\mathcal{T}_{i+1}(a, f) = \langle R_3, l_2 \rangle, \\ \qquad \mathit{where}\ \mathcal{T}_i(a, f) = \langle \_, l_2 \rangle \wedge R_1 \subseteq R_3 \\ \mathcal{A}_i \sqsubseteq l, l_1 \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \backslash \{(a, f) \mid a \in R\} \leq \mathcal{T}_{i+1} \backslash \{(a, f) \mid a \in R\} \\ \forall k \in \mathtt{region}(i).l \sqsubseteq \mathcal{A}_k \\ \mathcal{S}_i(0), \mathcal{S}_i(1) \in \mathtt{L}_I,\ \mathit{if}\ \mathcal{A}_i \in \mathtt{L}_I \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

$$\text{(T-GtFld)}\quad \frac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{getfield}\ f \\ \mathsf{Handler}(i) \downarrow \\ \mathcal{S}_i(0) = \langle R, l \rangle \\ \kappa = \bigsqcup_{a \in R} \mathcal{T}_i(a, f) \\ l \sqcup \kappa \sqcup \mathcal{A}_i \cdot (\mathcal{S}_i \backslash 0) \leq_{\mathcal{A}_i} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \\ \forall k \in \mathtt{region}(i).l \sqsubseteq \mathcal{A}_k \\ \mathcal{S}_i(0) \in \mathtt{L}_I,\ \mathit{if}\ \mathcal{A}_i \in \mathtt{L}_I \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

$$\text{(T-PtFldH)}\quad \frac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{putfield}\ f \\ \mathsf{Handler}(i) = i' \\ \mathcal{S}_i = \kappa' \cdot \langle R, l \rangle \cdot S, \text{ for some } S \\ a = \mathsf{Fresh}(\mathcal{T}_i) \\ \langle \{a\}, l \sqcup \mathcal{A}_i \rangle \cdot \epsilon \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\ \mathcal{V}_i \leq \mathcal{V}_{i'} \\ \mathcal{T}_i \oplus \{a \mapsto [f_1 : \mathtt{ft}_{\mathcal{A}_i}(f_1), \ldots, f_n : \mathtt{ft}_{\mathcal{A}_i}(f_n)]\} \leq \mathcal{T}_{i'} \\ \forall j \in \mathsf{Dom}(\mathcal{S}_i).\mathcal{A}_i \sqsubseteq \mathcal{S}_i(j) \\ \forall j \in \mathsf{Dom}(\mathcal{S}_i).\mathcal{S}_i(j) \in \mathtt{L}_I,\ \mathit{if}\ \mathcal{A}_i \in \mathtt{L}_I \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

$$\text{(T-GtFldH)}\quad \frac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{getfield}\ f \\ \mathsf{Handler}(i) = i' \\ \mathcal{S}_i = \langle R, l \rangle \cdot S, \text{ for some } S \\ a = \mathsf{Fresh}(\mathcal{T}_i) \\ \langle \{a\}, l \sqcup \mathcal{A}_i \rangle \cdot \epsilon \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\ \mathcal{V}_i \leq \mathcal{V}_{i'} \\ \mathcal{T}_i \oplus \{a \mapsto [f_1 : \mathtt{ft}_{\mathcal{A}_i}(f_1), \ldots, f_n : \mathtt{ft}_{\mathcal{A}_i}(f_n)]\} \leq \mathcal{T}_{i'} \\ \forall j \in \mathsf{Dom}(\mathcal{S}_i).\mathcal{A}_i \sqsubseteq \mathcal{S}_i(j) \\ \forall j \in \mathsf{Dom}(\mathcal{S}_i).\mathcal{S}_i(j) \in \mathtt{L}_I,\ \mathit{if}\ \mathcal{A}_i \in \mathtt{L}_I \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

$$\text{(T-PtFldNoH)}\quad \frac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{putfield}\ f \\ \mathsf{Handler}(i) \uparrow \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \kappa \cdot \langle R, l \rangle \cdot (\mathcal{S}_{err_i} \backslash 0) \\ \kappa \sqsubseteq \bigsqcap_{a \in R} \mathcal{T}_i(a, f) \\ a = \mathsf{Fresh}(\mathcal{T}_i) \\ \langle \{a\}, l \sqcup \mathcal{A}_i \rangle \sqsubseteq \mathcal{S}_{err_i}(0) \\ l \sqsubseteq \kappa_r \\ \mathcal{A}_i \sqsubseteq \kappa, l \\ \mathcal{V}_i \leq \mathcal{V}_{err_i} \\ \mathcal{T}_i \oplus \{a \mapsto [f_1 : \mathtt{ft}_{\mathcal{A}_i}(f_1), \ldots, f_n : \mathtt{ft}_{\mathcal{A}_i}(f_n)]\} \leq \mathcal{T}_{err_i} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

$$\text{(T-GtFldNoH)}\quad \frac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \mathtt{getfield}\ f \\ \mathsf{Handler}(i) \uparrow \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \langle R, l \rangle \cdot (\mathcal{S}_{err_i} \backslash 0) \\ \mathcal{S}_i(0) = \langle R, l \rangle \\ \kappa = \bigsqcup_{a \in R} \mathcal{T}_i(a, f) \\ a = \mathsf{Fresh}(\mathcal{T}_i) \\ \langle \{a\}, l \sqcup \mathcal{A}_i \rangle \sqsubseteq \mathcal{S}_{err_i}(0) \\ \mathcal{A}_i \sqsubseteq l \sqsubseteq \kappa_r \\ \mathcal{V}_i \leq \mathcal{V}_{err_i} \\ \mathcal{T}_i \oplus \{a \mapsto [f_1 : \mathtt{ft}_{\mathcal{A}_i}(f_1), \ldots, f_n : \mathtt{ft}_{\mathcal{A}_i}(f_n)]\} \leq \mathcal{T}_{err_i} \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

Fig. 4.3. Full set of typing schemes II

$$\text{(T-InvkVrtl)}$$
$$\dfrac{\begin{array}{l}
B[\overrightarrow{\bullet}](i) = \texttt{invokevirtual}\ m \\
\mathsf{Handler}_B(i) \downarrow \\
\mathsf{mt}_m(\kappa \sqcup \mathcal{A}_i) = \mathcal{A}'_1, \mathcal{T}'_1, \mathcal{T}'_{pf} \rhd ((\overrightarrow{x' : \kappa'}, \kappa'_r, E'), B') \\
\mathcal{A}_i \sqcup \kappa = \mathcal{A}'(1) \\
\mathcal{S}_i = \kappa'_1 \cdots \kappa'_n \cdot \kappa \cdot S,\ \text{for some } S \\
\mathcal{S}_i \leq_{\mathcal{A}_i} \kappa'_1 \cdots \kappa'_n \cdot \kappa \cdot (\mathcal{S}_{i+1} \backslash 0) \\
\mathcal{A}_i \sqsubseteq \kappa'_1, \cdots, \kappa'_n, \kappa \\
\kappa \sqcup \kappa'_r \sqcup \mathcal{A}_i \sqsubseteq \mathcal{S}_{i+1}(0) \\
\mathcal{V}_i \leq \mathcal{V}_{i+1} \\
\mathcal{T}_i \leq \mathcal{T}'_1 \leq \mathcal{T}'_{pf} \leq \mathcal{T}_{i+1} \\
\forall k \in \texttt{region}(i).\kappa \sqcup \kappa'_r \sqsubseteq \mathcal{A}_k,\ \text{if } i \in \mathsf{Dom}(B^\sharp) \\
\kappa'_1, \cdots, \kappa'_n, \kappa \in \mathtt{L}_I,\ \textit{if}\ \mathcal{A}_i \in \mathtt{L}_I
\end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

$$\text{(T-InvkVrtlH)}$$
$$\dfrac{\begin{array}{l}
B[\overrightarrow{\bullet}](i) = \texttt{invokevirtual}\ m \\
\mathsf{Handler}_B(i) = i' \\
\mathsf{mt}_m(\kappa \sqcup \mathcal{A}_i) = \mathcal{A}'_1, \mathcal{T}'_1, \mathcal{T}'_{pf} \rhd ((\overrightarrow{x' : \kappa'}, \kappa'_r, E'), B') \\
E' \neq \emptyset \\
\mathcal{A}_i \sqcup \kappa = \mathcal{A}'(1) \\
\kappa'_r \sqcup \mathcal{A}_i \cdot \epsilon \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\
\mathcal{T}_i \leq \mathcal{T}'_1 \leq \mathcal{T}'_{pf} \leq \mathcal{T}_{i'} \\
\mathcal{V}_i \leq \mathcal{V}_{i'} \\
\forall j \in \mathsf{Dom}(\mathcal{S}_i).\mathcal{A}_i \sqsubseteq \mathcal{S}_i(j) \\
\forall j \in \mathsf{Dom}(\mathcal{S}_i).\mathcal{S}_i(j) \in \mathtt{L}_I,\ \textit{if}\ \mathcal{A}_i \in \mathtt{L}_I
\end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

$$\text{(T-InvkVrtlNoH)}$$
$$\dfrac{\begin{array}{l}
B[\overrightarrow{\bullet}](i) = \texttt{invokevirtual}\ m \\
\mathsf{Handler}_B(i) \uparrow \\
\mathsf{mt}_m(\kappa \sqcup \mathcal{A}_i) = \mathcal{A}'_1, \mathcal{T}'_1, \mathcal{T}'_{pf} \rhd ((\overrightarrow{x' : \kappa'}, \kappa'_r, E'), B') \\
E' \neq \emptyset \\
\mathcal{A}_i \sqcup \kappa = \mathcal{A}'(1) \\
\mathcal{S}_i \leq_{\mathcal{A}_i} \kappa'_1 \cdots \kappa'_n \cdot \kappa \cdot (\mathcal{S}_{err_i} \backslash 0) \\
\kappa'_r \sqcup \mathcal{A}_i \sqsubseteq \mathcal{S}_{err_i}(0) \\
\mathcal{A}_i \sqcup \kappa \sqcup \kappa'_r \sqsubseteq \kappa_r \\
\mathcal{V}_i \leq \mathcal{V}_{err_i} \\
\mathcal{T}_i \leq \mathcal{T}'_1 \leq \mathcal{T}'_{pf} \leq \mathcal{T}_{err_i}
\end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd B[\overrightarrow{\bullet}]}$$

Fig. 4.4. Full set of typing schemes III

Now that the type system is in place and our motivating examples have been introduced we can formalize the notion of an active attack.

Definition 4.3 (Active attack). *A program a is an active attack under method type $(\overrightarrow{x : \kappa}, \kappa_r, E)$ if a*

- *does not contain `declassify` or `return` instructions,*
- *a handler is defined for each instruction that can raise an exception,*
- *all jumps to be local to a, and*
- *there exists a typing context $(\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T})$ such that the following judgement holds:*

$$\dfrac{\begin{array}{c}\mathcal{A}_{a_1} = (\bot_C, \mathtt{l}_\mathtt{A}) \quad \|\mathcal{S}_{a_1}\| = \|\mathcal{S}_{a_{exit}}\| \\ \forall i \in \mathsf{Dom}(a).\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, i \rhd ((\overrightarrow{x : \kappa}, \kappa_r, E), a)\end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T} \rhd ((\overrightarrow{x : \kappa}, \kappa_r, E), a)}$$

The index a_1 in $\mathcal{A}_{a_1}$ and $\mathcal{S}_{a_1}$ refers to the initial program point of a; and the index a_{exit} in $\mathcal{S}_{exit}$ refers to the program point $n + 1$ where n is the size of a. We assume that execution always ends at this program point. Also, as in the definition of well-typed programs, we assume the chosen $\mathcal{A}$ verifies $\forall i \in \mathsf{Dom}(a).\mathcal{A}_{a_1} \sqsubseteq \mathcal{A}_i$.

4.3.2 Type-Checking Examples

This section examines some examples to exhibit how our type system deals with them.

Example 4.4. Given the following instructions of a method:

```
1 load x
2 load y
3 declassify (L, H)
```

The typing rules generate the following constraints, with $\mathtt{A} = (\mathtt{L}, \mathtt{L})$:

i	$B(i)$	$\mathcal{V}_i(x)$	$\mathcal{V}_i(y)$	$\mathcal{S}_i$	$\mathcal{A}_i$	$Rule's \quad Precond.$	$Constraints$
1	`load` x	$(\mathtt{H},\mathtt{H})$	$(\mathtt{H},\mathtt{H})$	ϵ	$(\mathtt{L},\mathtt{H})$	$(\mathtt{H},\mathtt{H}) \cdot \epsilon \leq \beta_1 \cdot \epsilon$	$(\mathtt{H},\mathtt{H}) \sqsubseteq \beta_1$
2	`load` y	$(\mathtt{H},\mathtt{H})$	$(\mathtt{H},\mathtt{H})$	$\beta_1 \cdot \epsilon$	δ_2	$(\mathtt{H},\mathtt{H}) \cdot \beta_1 \cdot \epsilon \leq \beta_2 \cdot \beta_1 \cdot \epsilon \quad \delta_2 \sqsubseteq \beta_2$	$(\mathtt{H},\mathtt{H}) \sqsubseteq \beta_2 \quad \delta_2 \sqsubseteq \beta_2$
3	`declassify` $(\mathtt{L},\mathtt{H})$	$(\mathtt{H},\mathtt{H})$	$(\mathtt{H},\mathtt{H})$	$\beta_2 \cdot \beta_1 \cdot \epsilon$	δ_3	$\mathsf{I}(\delta_3), \mathsf{I}(\beta_2) \in \mathtt{H}_I$	$\mathsf{I}(\delta_3), \mathsf{I}(\beta_2) \in \mathtt{H}_I$

The constraints are satisfacible by:

$$\beta_1, \beta_2 = (\mathtt{H},\mathtt{H}) \qquad \delta_2, \delta_3 = (\mathtt{L},\mathtt{H})$$

If the attacker consists of a `pop` instruction, then declassification of the value of x is forced. If $[\bullet] = $ `pop` and this code is inserted between the instruction 2 and 3:

```
1 load x
2 load y
  [•]
3 declassify (L, H)
```

In this case, the constraint $\mathsf{I}(\mathcal{S}(0)) \in \mathtt{L}_I$ of `pop` instruction is violated. So this is not a possible attack.

We note that if the attacker inserts the code `goto` 4 or `return` between the instruction 2 and 3 then by affecting the control-flow y is not declassified. However, these code fragments violate the definition of valid attacks. That is, these attacks are not allowed.

Now, suppose the attacker reuses y with a low-integrity value, such as in $[\bullet] = $ `load` $z_{(\mathtt{H},\mathtt{L})}$; `store` y and this code is inserted between the instruction 1 and 2. Then in line 3 a declassification of z is attempted (a low-integrity value). In this case, the constraint $\mathsf{I}(\mathcal{V}(y)) \in \mathtt{L}_I$ of `store` instruction is violated. So this is not a possible attack.

4.4 Soundness

The type system enforces two interesting properties: *noninterference* and *robust declassification*. Recall that noninterference states that any two (terminating) runs of a well-typed method, that does not use `declassify`, starting from initial states that coincide on public input data produce final states in which public output data are identical. This holds immediately from the results of Section 3.6.1; we merely restate the definition and statement of the noninterference result. Robust declassification states that an attacker may not manipulate the declassification mechanism to leak more information than intended. This property applies to programs that may contain `declassify` instructions. Section 4.4.2 introduces the precise definition and proves the main result of this Chapter.

4.4.1 Noninterference

Indistinguishability for states at some level λ is formulated by considering each of its components at a time (values, local variable assignments, heaps and stacks) in the same way as Chapter 3. States are declared indistinguishable depending on what may be observed and this, in turn, depends on the level κ of the observer. Henceforth we turn to the definitions of noninterference (Definition 3.41) presented in the Chapter 3.

Proposition 4.5. *All well-typed methods $M[\overrightarrow{a}]$, without* $\texttt{declassify}$ *instructions, satisfy* $\mathsf{NI}(M[\overrightarrow{a}])$.

Proposition 4.5 is a consequence of a stronger property (Proposition 4.10) that we state below. This latter property is used in our result on robustness (Proposition 4.12). First we introduce some definitions.

Definition 4.6 (Trace). *The trace of execution of B from a state $\langle i_0, \alpha, \sigma, \eta\rangle$ for B is denoted $\mathsf{Tr}_B(\langle i_0, \alpha, \sigma, \eta\rangle)$ and defined to be the sequence $\langle i_0, \alpha, \sigma, \eta\rangle \longrightarrow_B \langle i_1, \alpha_1, \sigma_1, \eta_1\rangle \longrightarrow_B \langle i_2, \alpha_2, \sigma_2, \eta_2\rangle \longrightarrow_B \cdots$.*

Definition 4.7. *Let the sequence states $t_1 = s_{i_1} \cdots s_{i_n}$ and $t_2 = s_{j_1} \cdots s_{j_n}$, $\beta_1 \vdash t_1 \sim^\lambda t_2$ holds iff for all $k \in 1..n$, $\beta_k \vdash s_{i_k} \sim^\lambda s_{j_k}$, for some $\beta_2 \cdots \beta_n$ such that $\beta_1 \subseteq \beta_2 \subseteq \cdots \subseteq \beta_n$.*

We recall that if $s = \langle i, \alpha, \sigma, \eta\rangle$ then we say $\mathtt{pc}(s) = i$ and $\mathcal{A}_s = \mathcal{A}_i$.

Because computation steps can be observed at level λ we identify traces formed only by states with security level less than or equal to lambda

Definition 4.8. *The λ-projection of a trace t, written $t|_\lambda$, where*

$$t = s_{i_0} \longrightarrow_{B[\overrightarrow{a}]} s_{i_1} \longrightarrow_{B[\overrightarrow{a}]} s_{i_2} \longrightarrow_{B[\overrightarrow{a}]} s_{i_3} \longrightarrow_{B[\overrightarrow{a}]} s_{i_4} \cdots$$

is the sequence of states $s_{j_0} s_{j_1} s_{j_2} s_{j_3} \cdots$ s.t. $j_0 < j_1 < j_2 < \cdots$ and for all i_k, if $\mathcal{A}(i_k) \sqsubseteq \lambda$ then $i_k = \mathtt{pc}(j_n)$, for some n.

By example, if $t = s_1 \longrightarrow s_2 \longrightarrow s_3 \longrightarrow s_4 \longrightarrow s_5$, $\mathcal{A}(1), \mathcal{A}(3), \mathcal{A}(4) \sqsubseteq \lambda$ and $\mathcal{A}(2), \mathcal{A}(5) \not\sqsubseteq \lambda$ then $t|_\lambda = s_1 \longrightarrow s_3 \longrightarrow s_4$.

Definition 4.9 (Trace Indistinguishability). *Let t_1 and t_2 two finite traces of the same length are indistinguishable, written $\beta \vdash t_1 \sim^\lambda t_2$, iff their λ-projections are indistinguishable $\beta \vdash t_1|_\lambda \sim^\lambda t_2|_\lambda$.*

Proposition 4.10. *Suppose $M[\overrightarrow{a}]$ has no* $\texttt{declassify}$ *instructions and s_1 and s_2 are initial states such that $\beta \vdash s_1 \sim^\lambda s_2$. Then $\beta \vdash \mathsf{Tr}_{B[\overrightarrow{a}]}(s_1) \sim^\lambda \mathsf{Tr}_{B[\overrightarrow{a}]}(s_2)$.*

Proof. By Proposition 3.42 the result follows. ∎

4.4.2 Robustness

We show that typable programs satisfy robustness.

Definition 4.11 (Robustness). *Let $M[\overrightarrow{\bullet}] = ((\overline{x : \kappa}, \kappa_r, E), B[\overrightarrow{\bullet}])$ be a well-typed method and β a location bijection set. We say that it satisfies robustness with respect to active attacks at level $\mathtt{A}$ if for every s_1, s_1' initial states and $\overrightarrow{a_1}$, $\overrightarrow{a_2}$ active attacks:*

$$\beta \vdash \mathsf{Tr}_{B[\overrightarrow{a_1}]}(s_1) \sim^{(\mathtt{C(A)}, \top_I)} \mathsf{Tr}_{B[\overrightarrow{a_1}]}(s_1') \text{ implies } \beta \vdash \mathsf{Tr}_{B[\overrightarrow{a_2}]}(s_1) \sim^{(\mathtt{C(A)}, \top_I)} \mathsf{Tr}_{B[\overrightarrow{a_2}]}(s_1')$$

Robust declassification holds if the attacker's observations from the execution of $M[\overrightarrow{a_2}]$ may not reveal any secrets apart from what the attacker already knows from observations about the execution of $M[\overrightarrow{a_1}]$. The main result of this work is therefore:

Proposition 4.12. *All well-typed methods satisfy robustness.*

In order to address the proof two auxiliary results are required. The first (Lemma 4.13) states that an active attack cannot consult or modify high-integrity data on the stack and the second (Lemma 4.23) extends this observation to the states of a trace.

Proposition 4.13. *An active attack a:*

(i) cannot consult or modify high-integrity data on the stack.

(ii) satisfies noninterference.

Proof. The second item follows immediately from Proposition 3.42. The first one is proved by case analysis on the instructions of a. Before proceeding, we recall from Definition 4.3, $\mathcal{A}_1 = (\bot_C, \mathsf{l}_{\mathtt{A}})$ and $\mathcal{A}_1 \sqsubseteq \mathcal{A}_i$ for all program points i of a. Therefore, for all program points i of a $(\bot_C, \mathsf{l}_{\mathtt{A}}) \sqsubseteq \mathcal{A}_i$. We supply two sample cases. The remaining cases are similar.

- Case `load` x. As mentioned $(\bot_C, \mathsf{l}_{\mathtt{A}}) \sqsubseteq \mathcal{A}_i$. Also, since a is well-typed, $\mathcal{V}_i(x) \sqcup \mathcal{A}_i \cdot \mathcal{S}_i \leq \mathcal{S}_{i+1}$. By transitivity $(\bot_C, \mathsf{l}_{\mathtt{A}}) \sqsubseteq \mathcal{S}_{i+1}(0)$, hence $\mathsf{l}_{\mathtt{A}} \sqsubseteq \mathsf{l}(\mathcal{S}_{i+1}(0))$. But, since $\mathtt{H}_I = \{\kappa \mid \mathsf{l}_{\mathtt{A}} \not\sqsubseteq \mathsf{l}_\kappa\}$, $\mathsf{l}(\mathcal{S}_{i+1}(0)) \notin \mathtt{H}_I$.
- Case `store` x. As mentioned $(\bot_C, \mathsf{l}_{\mathtt{A}}) \sqsubseteq \mathcal{A}_i$. Since a is well-typed, $\mathcal{A}_i \sqsubseteq \mathcal{S}_i(0)$ and therefore by transitivity $(\bot_C, \mathsf{l}_{\mathtt{A}}) \sqsubseteq \mathcal{S}_i(0)$ and hence $\mathsf{l}_{\mathtt{A}} \sqsubseteq \mathsf{l}(\mathcal{S}_i(0))$. But, since $\mathtt{H}_I = \{\kappa \mid \mathsf{l}_{\mathtt{A}} \not\sqsubseteq \mathsf{l}_\kappa\}$, $\mathsf{l}(\mathcal{S}_i(0)) \notin \mathtt{H}_I$ (1). Furthermore, a well-typed implies $\mathcal{S}_i(0) \sqsubseteq \mathcal{V}_{i+1}(x)$. Then $\mathsf{l}(\mathcal{V}_{i+1}(x)) \notin \mathtt{H}_I$, by (1) and transitivity of $\sqsubseteq$.

∎

First, we defined the indistinguishability for integrity values. States are declared indistinguishable depending on what may be modified and this, in turn, depends on the security level l of the observer. The following definitions are similar to indistinguishability definitions for confidentiality (Section 3.6.1).

Definition 4.14 (Value Integrity Indistinguishability). *Given values v_1, v_2, security levels $l, \mathtt{A}$ ($\mathtt{A}$ is the attacker security level) and β a location bijection set we define $\beta, l \vdash v_1 \simeq^{\mathtt{A}} v_2$ ("values v_1 and v_2 are indistinguishable at level $\mathtt{A}$ w.r.t. observer level l") as follows:*

$$\text{(L-NULL-I)}\ \frac{l \in \mathtt{H}_I}{\beta, l \vdash \mathit{null} \simeq^{\mathtt{A}} \mathit{null}} \qquad \text{(L-INT-I)}\ \frac{v \in \mathbb{Z} \qquad l \in \mathtt{H}_I}{\beta, l \vdash v \simeq^{\mathtt{A}} v}$$

$$\text{(L-LOC-I)}\ \frac{v_1, v_2 \in \mathbb{L} \quad l \in \mathtt{H}_I \quad \beta(v_1) = v_2}{\beta, l \vdash v_1 \simeq^{\mathtt{A}} v_2} \qquad \text{(H-VAL-I)}\ \frac{l \notin \mathtt{H}_I}{\beta, l \vdash v_1 \simeq^{\mathtt{A}} v_2}$$

We remember that the $\mathtt{H}_I$ is determined by the attacker security level $\mathtt{A}$.

Definition 4.15 (Local Variable Array Integrity Indistinguishability). *Let α_1, α_2 be local variable arrays, V_1, V_2 frame types, β a location bijection set and $l, \mathtt{A}$ a security levels. We write $\beta, (V_1, V_2), l \vdash \alpha_1 \simeq^{\mathtt{A}} \alpha_2$ (or $\beta, V_1, l \vdash \alpha_1 \simeq^{\mathtt{A}} \alpha_2$ when $V_1 = V_2$ and $l \in \mathtt{H}_I$).*

- *Low-indist. Local Variable ($l \in \mathtt{H}_I$ and $V_1 = V_2$). α_1 and α_2 are considered low-indist. at level $\mathtt{A}$ if for all $x \in \mathbb{X}$,*
$$\beta, \lfloor V_1(x) \rfloor \vdash \alpha_1(x) \simeq^{\mathtt{A}} \alpha_2(x).$$
- *High-indist. Local Variable ($l \notin \mathtt{H}_I$). α_1 and α_2 are considered high-indist. at level $\mathtt{A}$ if for all $x \in \mathbb{X}$,*
$$\beta, \lfloor V_1(x) \sqcup V_2(x) \rfloor \vdash \alpha_1(x) \simeq^{\mathtt{A}} \alpha_2(x).$$

Definition 4.16 (Heap Integrity Indistinguishability). *Let η_1, η_2 be heaps, T_1, T_2 heap types, β a location bijection set and $\mathtt{A}$ security level. We define η_1 and η_2 to be indistinguishable at T_1, T_2 under β (we write $\beta, (T_1, T_2) \vdash \eta_1 \simeq^{\mathtt{A}} \eta_2$ or $\beta, T_1 \vdash \eta_1 \simeq^{\mathtt{A}} \eta_2$ in the case that $T_1 = T_2$) if:*

1. *β is well-defined w.r.t. η_1, η_2, T_1 and T_2, and*
2. *for every $o \in \mathsf{Dom}(\beta)$, for every field name $f \in \mathsf{Dom}(\eta_1(o))$*
$$\beta, \lfloor T_1(\beta^{\lhd -1}(o), f) \rfloor \vdash \eta_1(o, f) \simeq^{\mathtt{A}} \eta_2(\beta(o), f).$$

Definition 4.17 (Stack Integrity Indistinguishability). *Let σ_1, σ_2 be stacks, S_1, S_2 stack types, β a location bijection set and $l, \mathtt{A}$ security levels. We write $\beta, (S_1, S_2), l \vdash \sigma_1 \simeq^{\mathtt{A}} \sigma_2$ (or $\beta, S_1, l \vdash \sigma_1 \simeq^{\mathtt{A}} \sigma_2$, when $S_1 = S_2$ and $l \in \mathtt{H}_I$).*

- *Low-indist. stacks ($l \in \mathtt{H}_I$).*

$$\text{(L-Nil-I)}\ \frac{l \in \mathtt{H}_I}{\beta, \epsilon, l \vdash \epsilon \simeq^{\mathtt{A}} \epsilon} \qquad \text{(Cons-L-I)}\ \frac{\beta, S, l \vdash \sigma_1 \simeq^{\mathtt{A}} \sigma_2 \quad \beta, \lfloor \kappa \rfloor \vdash v_1 \simeq^{\mathtt{A}} v_2 \quad l \in \mathtt{H}_I}{\beta, \kappa \cdot S, l \vdash v_1 \cdot \sigma_1 \simeq^{\mathtt{A}} v_2 \cdot \sigma_2}$$

- *High-indist. stacks ($l \notin \mathtt{H}_I$).*

$$\text{(H-Low-I)}\ \frac{\beta, S, l \vdash \sigma_1 \simeq^{\mathtt{A}} \sigma_2 \quad l \in \mathtt{H}_I \quad l' \notin \mathtt{H}_I}{\beta, (S, S), l' \vdash \sigma_1 \simeq^{\mathtt{A}} \sigma_2}$$

$$\text{(H-Cons-L-I)}\ \frac{\beta, (S_1, S_2), l \vdash \sigma_1 \simeq^{\mathtt{A}} \sigma_2 \quad l, \lfloor \kappa \rfloor \notin \mathtt{H}_I}{\beta, (\kappa \cdot S_1, S_2), l \vdash v \cdot \sigma_1 \simeq^{\mathtt{A}} \sigma_2} \qquad \text{(H-Cons-R-I)}\ \frac{\beta, (S_1, S_2), l \vdash \sigma_1 \simeq^{\mathtt{A}} \sigma_2 \quad l, \lfloor \kappa \rfloor \notin \mathtt{H}_I}{\beta, (S_1, \kappa \cdot S_2), l \vdash \sigma_1 \simeq^{\mathtt{A}} v \cdot \sigma_2}$$

Definition 4.18 (Machine State Integrity Indistinguishability). *Given security level $\mathtt{A}$, β a location bijection set, $\beta \vdash \langle i, \alpha, \sigma, \eta \rangle \simeq^{\mathtt{A}} \langle i', \alpha', \sigma', \eta' \rangle$ holds iff*

1. *$\mathcal{A}_i, \mathcal{A}_{i'} \notin \mathtt{H}_I$ or ($\mathcal{A}_i = \mathcal{A}_{i'} \in \mathtt{H}_I$ and $i = i'$);*
2. *$\beta, (\mathcal{V}_i, \mathcal{V}_{i'}), \mathcal{A}_i \vdash \alpha \simeq^{\mathtt{A}} \alpha'$;*
3. *$\beta, (\mathcal{S}_i, \mathcal{S}_{i'}), \mathcal{A}_i \vdash \sigma \simeq^{\mathtt{A}} \sigma'$; and*
4. *$\beta, (\mathcal{T}_i, \mathcal{T}_{i'}) \vdash \eta \simeq^{\mathtt{A}} \eta'$.*

Definition 4.19 (Final State Indistinguishability). *Given security level $\mathtt{A}$, β a location bijection set, $\beta \vdash \langle p_f, v, \eta \rangle \simeq^{\mathtt{A}} \langle p_f, v', \eta' \rangle$ holds if the following hold:*

1. *$\beta, \lfloor \kappa_r \rfloor \vdash v \simeq^{\mathtt{A}} v'$; and*
2. *$\beta, \mathcal{T}_{p_f} \vdash \eta \simeq^{\mathtt{A}} \eta'$.*

The three unwinding lemmas follow together with the statement of Lemma 4.23 **Preservation of Equality of High Integrity Data** and its proof. For proofs of the unwinding lemmas consult Appendix B.3. The Lemma 4.23 says that, assuming that in the initial states agree on high-integrity data then they also agree on high-integrity data at the intermediate states. This is a form of noninterference of high-integrity values.

Lemma 4.20 (One-Step Preservation of Equality of High Integrity Data on Low Context). *Suppose*

1. *$\mathcal{A}_{s_1} \in \mathtt{H}_I$ and $\mathtt{pc}(s_1) = \mathtt{pc}(s_2)$*
2. *$s_1 \longrightarrow_{B[\overrightarrow{a_1}]} s_1'$;*
3. *$s_2 \longrightarrow_{B[\overrightarrow{a_2}]} s_2'$; and*
4. *$\beta \vdash s_1 \simeq^{\mathtt{A}} s_2$ for some location bijection set β.*

Then
$\beta' \vdash s_1' \simeq^{\mathtt{A}} s_2'$ and $\mathtt{pc}(s_1') = \mathtt{pc}(s_2')$, for some $\beta' \supseteq \beta$.

Lemma 4.21 (One-Step Preservation of Equality of High Integrity Data on High Context). *Let $\mathtt{pc}(s_1), \mathtt{pc}(s_1'), \mathtt{pc}(s_2) \in \mathtt{L}_I$. Furthermore, suppose:*

1. *$\beta \vdash s_1 \simeq^{\mathtt{A}} s_1'$ for some location bijection set β;*
2. *$s_1 \longrightarrow_{B[\overrightarrow{a}]} s_2$.*

Then $\beta \vdash s_2 \simeq^{\mathtt{A}} s'_1$.

Lemma 4.22 (One-Step Preservation of Equality of High Integrity Data on High to Low Context). *Let* $\mathtt{pc}(s_1), \mathtt{pc}(s_2) \in \mathtt{L}_I$ *and* $\mathtt{pc}(s'_1), \mathtt{pc}(s'_2) \in \mathtt{H}_I$. *Furthermore, suppose:*

1. $s_1 \longrightarrow_{B[\overrightarrow{a_1}]} s'_1$*;*
2. $s_2 \longrightarrow_{B[\overrightarrow{a_2}]} s'_2$*;*
3. $\beta \vdash s_1 \simeq^{\mathtt{A}} s_2$*, for some location bijection set* β*; and*
4. $\mathtt{pc}(s'_1) = \mathtt{pc}(s'_2)$.

Then $\beta \vdash s'_1 \simeq^{\mathtt{A}} s'_2$.

Lemma 4.23 (Preservation of Equality of High Integrity Data). *Let* $M[\overrightarrow{\bullet}] = ((\overline{x : \kappa}, \kappa_r, E), B[\overrightarrow{\bullet}])$ *be a well-typed method,* $\overrightarrow{a_1}$*,* $\overrightarrow{a_2}$ *active attacks,* $\mathtt{A}$ *attacker security level,* s_1, s'_1 *initial states,* β *a location bijection set and*

1. $\beta \vdash s_1 \simeq^{(\top_C, l)} s'_1$*, where for all* l' *in* $\mathtt{H}_I$ $l' \sqsubseteq l$ *and* l *in* $\mathtt{H}_I$*;*
2. $s_1, \cdots, s_n$ *in* $\mathsf{Tr}_{B[\overrightarrow{a_1}]}(s_1)|_{(\top_C, l)}$*; and*
3. $s'_1 \cdots, s'_n$ *in* $\mathsf{Tr}_{B[\overrightarrow{a_2}]}(s'_1)|_{(\top_C, l)}$.

Then for all i *in* $\{1, \cdots, n\}$ $\beta' \vdash s_i \simeq^{(\top_C, l)} s'_i$*, for some* $\beta' \supseteq \beta$.

Proof. Consider any terminating execution path of $M[\overrightarrow{a_1}]$ and of $M[\overrightarrow{a_2}]$:

$$\begin{array}{l} s_1 \longrightarrow_{B[\overrightarrow{a_1}]} s_2 \longrightarrow_{B[\overrightarrow{a_1}]} \cdots \longrightarrow_{B[\overrightarrow{a_1}]} s_n \\ s'_1 \longrightarrow_{B[\overrightarrow{a_2}]} s'_2 \longrightarrow_{B[\overrightarrow{a_2}]} \cdots \longrightarrow_{B[\overrightarrow{a_2}]} s'_m \end{array}$$

where, $\beta \vdash s_1 \simeq^{(\top_C, l)} s'_1$. We proceed as follows. Starting from s_1 and s'_1, repeatedly apply Lemma 4.20 until it is no longer possible. Let s_j and s'_j be the states that are reached. We have two cases to treat:

1. Case $j = n$. Suppose $s_j = s_n$ and $s'_j = s'_m$. Then these states are final states and by Lemma 4.20, $\beta \vdash s_n \simeq^{(\top_C, l)} s'_m$. We note that for all intermediate states, s_j and s'_j, $\beta \vdash s_j \simeq^{(\top_C, l)} s'_j$ holds, by Lemma 4.20. Furthermore, $\mathcal{A}_{s_j}, \mathcal{A}_{s'_j} \in \mathtt{H}_I$ then s_j in $\mathsf{Tr}_{B[\overrightarrow{a_1}]}(s_1)|_{(\top_C, l)}$ and s'_j in $\mathsf{Tr}_{B[\overrightarrow{a_2}]}(s'_1)|_{(\top_C, l)}$.
2. Case $j < n$. $\mathcal{A}_{s_j}, \mathcal{A}_{s'_j} \in \mathtt{L}_I$. As a consequence, the program counter points to the attackers' code. Before proceeding, we recall from Sec. 4.3.1, $\mathtt{pc}(s_j) = (\bot_C, \mathsf{l}_{\mathtt{A}}) = \mathtt{pc}(s'_j)$ and $\mathtt{pc}(s_j) \sqsubseteq \mathcal{A}_i$ for all program points i of a_1 and a_2. Therefore, for all program points i of a_1 and a_2, $(\bot_C, \mathsf{l}_{\mathtt{A}}) \sqsubseteq \mathcal{A}_i$ and hence $\mathcal{A}_i \in \mathtt{L}_I$. Since the hypothesis of Lemma 4.21 is satisfied we now repeatedly apply Lemma 4.21 in both branches of the executions until it is no longer possible and, say, $s_{1_{exit}}$ and $s'_{2_{exit}}$ are the reached states. Then $\beta \vdash s_{1_{exit}} \simeq^{(\top_C, l)} s'_{2_{exit}}$.
By Lemma 4.21, $\mathcal{A}_{s_{1_{exit}}}, \mathcal{A}_{s'_{2_{exit}}} \in \mathtt{L}_I$ and we know that $1_{exit} \mapsto h_1$, $2_{exit} \mapsto h_2$ and $h_1 = h_2$ and $\mathcal{A}_{h_1}, \mathcal{A}_{h_2} \in \mathtt{H}_I$. Then, by Lemma 4.22, we can affirm that $\beta \vdash s_{h_1} \simeq^{(\top_C, l)} s'_{h_2}$. Furthermore, we have that s_{h_1} in $\mathsf{Tr}_{B[\overrightarrow{a_1}]}(s_1)|_{(\top_C, l)}$ and s'_{h_2} in $\mathsf{Tr}_{B[\overrightarrow{a_2}]}(s'_1)|_{(\top_C, l)}$.

We repeat the argument, namely applying Lemma 4.20, cases 1 and case 2 until no longer possible. Given that the derivations are finite, eventually case 1 is reached. ∎

4.4.3 Robustness Proof

We now deliver the promised result, namely the proof of Proposition 4.12.

Proposition 4.12 All well-typed methods satisfy robustness.

Proof. Consider any two execution paths of $M[\overrightarrow{a_2}] = ((\overline{x : \kappa}, \kappa_r, E), B[\overrightarrow{a_2}])$:

$$\begin{array}{l} s_1 \longrightarrow_{B[\overrightarrow{a_2}]} s_2 \longrightarrow_{B[\overrightarrow{a_2}]} \cdots \longrightarrow_{B[\overrightarrow{a_2}]} s_n \\ s'_1 \longrightarrow_{B[\overrightarrow{a_2}]} s'_2 \longrightarrow_{B[\overrightarrow{a_2}]} \cdots \longrightarrow_{B[\overrightarrow{a_2}]} s'_m \end{array}$$

where

1. $s_1 = \langle 1, \alpha_1, \sigma_1, \eta_1 \rangle$ and $s'_1 = \langle 1, \alpha'_1, \sigma'_1, \eta'_1 \rangle$ with $\sigma_1 = \sigma'_1 = \epsilon$,
2. $s_n = \langle p_f, v, \eta \rangle$ and $s'_m = \langle p_f, v', \eta' \rangle$, and
3. $\beta \vdash s_1 \sim^{(\mathtt{C(A)}, \top_I)} s'_1$.

And suppose $\mathcal{A}_1 \in \mathtt{L}_C$.

If $\mathtt{H}_I = \emptyset$ then declassification is disallowed by the typing rules and the result follows from Proposition 4.10. Otherwise we proceed as follows. Starting from s_1 and s'_1 repeatedly apply Lemma 3.44 until it is no longer possible. Let s_{j_1} and s'_{j_2} be the states that are reached. Note $\beta' \vdash s_{j_1} \sim^{(\mathtt{C(A)}, \top_I)} s'_{j_2}$. If $j = n$, then the result holds immediately since the full trace has been attained. Otherwise, $j < n$ and we have two cases to treat (we write $\mathcal{A}_s$ for $\mathcal{A}_i$, where $s = \langle i, \alpha, \sigma, \eta \rangle$):

1. Case $\mathcal{A}_{s_{j_1}}, \mathcal{A}_{s'_{j_2}} \sqsubseteq (\mathtt{C(A)}, \top_I)$. Let $s_{j_1} = \langle i, \alpha, v \cdot \sigma, \eta \rangle$ and $s'_{j_2} = \langle i', \alpha', v' \cdot \sigma', \eta' \rangle$ and $B[\overrightarrow{a_2}](i) =$ `declassify` κ. From $\beta' \vdash s_{j_1} \sim^{(\mathtt{C(A)}, \top_I)} s'_{j_2}$ and $\mathcal{A}_{s_{j_1}}, \mathcal{A}_{s'_{j_2}} \sqsubseteq (\mathtt{C(A)}, \top_I)$ we deduce $i = i'$. Also, from the semantics $s_{j_1+1} = \langle i+1, \alpha, v \cdot \sigma, \eta \rangle$ and $s'_{j_2+1} = \langle i+1, \alpha', v' \cdot \sigma', \eta' \rangle$. Finally, by typability of $B[\overrightarrow{\bullet}]$ all hypothesis of T-Declassify are assumed.
   We wish to prove $\beta' \vdash s_{j_1+1} \sim^{(\mathtt{C(A)}, \top_I)} s'_{j_2+1}$. The only non-trivial case is $\beta', \mathcal{S}_{i+1}, \mathcal{A}_{i+1} \vdash v \cdot \sigma \sim^{(\mathtt{C(A)}, \top_I)} v' \cdot \sigma'$
   For this we first check that $\beta', \mathcal{S}_{i+1} \backslash 0, \mathcal{A}_{i+1} \vdash \sigma \sim^{(\mathtt{C(A)}, \top_I)} \sigma'$. This fact is obtained by reasoning as follows:
   - If $\mathcal{A}_{i+1} \sqsubseteq (\mathtt{C(A)}, \top_I)$, by $\beta', \mathcal{S}_i \backslash 0, \mathcal{A}_i \vdash \sigma \sim^{(\mathtt{C(A)}, \top_I)} \sigma'$. $(\mathcal{S}_i \backslash 0) \leq_{\mathcal{A}_i} (\mathcal{S}_{i+1} \backslash 0)$ and Lemma A.6, we have that $\beta', \mathcal{S}_{i+1} \backslash 0, \mathcal{A}_i \vdash \sigma \sim^{(\mathtt{C(A)}, \top_I)} \sigma'$. Then the result follows from Lemma A.9.
   - If $\mathcal{A}_{i+1} \not\sqsubseteq (\mathtt{C(A)}, \top_I)$, the result follows from the previous item and H-Low.

   Second we must check that $\beta', \lfloor \mathcal{S}_{i+1}(0) \rfloor \vdash v \sim^{(\mathtt{C(A)}, \top_I)} v'$. By $\beta' \vdash s_{j_1} \sim^{(\mathtt{C(A)}, \top_I)} s'_{j_2}$, we know that $\beta', \lfloor \mathcal{S}_i(0) \rfloor \vdash v \sim^{(\mathtt{C(A)}, \top_I)} v'$. We consider two cases depending on whether $\mathcal{S}_i(0) \sqsubseteq (\mathtt{C(A)}, \top_I)$ or $\mathcal{S}_i(0) \not\sqsubseteq (\mathtt{C(A)}, \top_I)$.
   Suppose $\mathcal{S}_i(0) \sqsubseteq (\mathtt{C(A)}, \top_I)$.
   - if $\mathcal{S}_{i+1}(0) \sqsubseteq (\mathtt{C(A)}, \top_I)$, $\beta', \lfloor \mathcal{S}_{i+1}(0) \rfloor \vdash v \sim^{(\mathtt{C(A)}, \top_I)} v'$ follows from $\beta' \vdash s_{j_1} \sim^{(\mathtt{C(A)}, \top_I)} s'_{j_2}$ and value indist.;
   - otherwise, we conclude directly by value indist.

   Suppose now $\mathcal{S}_i(0) \not\sqsubseteq (\mathtt{C(A)}, \top_I)$. If $v = v'$ then $\beta', \lfloor \mathcal{S}_{i+1}(0) \rfloor \vdash v \sim^{(\mathtt{C(A)}, \top_I)} v'$ is immediate. But, if $v \neq v'$ and $\mathcal{S}_{i+1}(0) \sqsubseteq (\mathtt{C(A)}, \top_I)$, since declassification has occurred, indistinguishability is not a priori guaranteed.
   Since v, v' are high-integrity data and active attackers cannot modify them (Proposition 4.13(i)), these same values are at the same program point in the execution of $B[\overrightarrow{a_1}]$ (Lemma 4.23). Then these same data, v, v', are also declassified by $B[\overrightarrow{a_1}]$ at this program point (since `declassify` instructions do not belong to $B[\overrightarrow{\bullet}]$). By Lemma 4.23, if high integrity values not are equal at the same program point, then there exist high integrity data on initial states s.t. they are not equal either. If $v \neq v'$, then the given assumption $\beta \vdash \mathsf{Tr}_{B[\overrightarrow{a_1}]}(s_1) \sim^{(\mathtt{C(A)}, \top_I)} \mathsf{Tr}_{B[\overrightarrow{a_1}]}(s'_1)$ would fail. Hence it must be the case that $v = v'$.
2. Case $\mathcal{A}_{s_{j_1}}, \mathcal{A}_{s'_{j_2}} \not\sqsubseteq (\mathtt{C(A)}, \top_I)$.
   As a consequence, there exists a program point k in $\mathsf{Dom}(B)^\sharp$ such that k was the value of the program counter for s_{j_1-1} and s'_{j_2-1}. Furthermore, let $\mathtt{pc}(s_{j_1}) = g$ and $\mathtt{pc}(s'_{j_2}) = g'$ then we have $k \mapsto g$ and $k \mapsto g'$ by the definition of the Successor Relation. By the SOAP properties (3.10(i)), both $g, g' \in \mathsf{region}(k)$.
   Furthermore, $\forall j \in \mathsf{region}(k).\mathcal{A}_j \not\sqsubseteq (\mathtt{C(A)}, \top_I)$ follows from the conditions of the typing rule T-If, T-GtFld, T-PtFld or T-InvkVrtl. Therefore the hypothesis of Lemma 3.45 is satisfied.

Now, we repeatedly apply Lemma 3.45 in both branches of executions until it is no longer possible. Let us call s_{h_1} and s'_{h_2} the reached states. By transitivity and symmetry of indistinguishability of states, (Lemma 3.40), $\beta' \vdash s_{h_1} \sim^{(\mathtt{C}(\mathtt{A}),\top_I)} s'_{h_2}$. By Lemma 3.45, $\mathcal{A}_{s_{h_1}}, \mathcal{A}_{s'_{h_2}} \not\sqsubseteq (\mathtt{C}(\mathtt{A}),\top_I)$. Also, if $h_1 \mapsto h'_1$ and $h_2 \mapsto h'_2$, then $\mathcal{A}_{h'_1}, \mathcal{A}_{h'_2} \sqsubseteq (\mathtt{C}(\mathtt{A}),\top_I)$. We are therefore at a junction point, Lemma A.12, $h'_1 = \mathsf{jun}(k) = h'_2$. Finally, by Lemma 3.46, $\beta' \vdash s_{h'_1} \sim^{(\mathtt{C}(\mathtt{A}),\top_I)} s'_{h'_2}$.

We repeat the argument, namely applying Lemma 3.44 until no longer possible and then analyzing cases 1 and case 2. Given that the derivations are finite, eventually final states are reached. ▪

4.5 Discussion

We present a type system for ensuring secure information flow in a core JVM-like language that includes a mechanism for performing downgrading of confidential information. It is proved that the type system enforces robustness of the declassification mechanism in the sense of [127]: attackers may not affect what information is released or whether information is released at all. The restriction of the attackers ability to modify data is handled by enriching the confidentiality lattice with integrity levels.

In presence of flow-sensitive type systems we have showed that variable reuse endows the attacker with further means to affect declassification than those available in high-level languages in which variables are assigned a fixed type during their entire lifetime. Additional means of enhancing the ability of an attacker to declassify information in stack-based languages (e.g. bytecode) is by manipulating the stack, as exemplified in Sec. 4.1. Also, in unstructured low-level languages, we are required to restrict the use of jumps.

We note that the infrastructure developed for the type system of Chapter 3 (typing schemes, definition of indistinguishability, lemmas, theorems and proofs) was used in this Chapter to show that well-typed programs satisfy robustness.

4.5.1 Extensions

The inclusion of a primitive for *endorsement* (for upgrading the integrity of data) as studied elsewhere [85, 67] in the setting of high-level languages would be interesting. Note that although confidentiality and integrity are dual properties, the treatment of `declassify` and `endorse` should not be symmetric for otherwise the attacker would be given total control over the data. We anticipate no major difficulties in adapting *qualified robustness* [85] in our type system. Although, qualified robustness is in some cases more permissive than desired [85] and we must probe that the type system meets *posibilistic* non-interference.

4.5.2 Related work

A brief description of related literature on type based information flow analysis for declassification and for low-level languages follows. A recent survey of research on declassification techniques is presented by Sabelfeld and Sands [102].

Zdancewic and Myers [127] introduce robust declassification (RD) in an attempt to capture the desirable restrictions on declassification. Robustness addresses an important issue for security analysis: the possibility that an attacker can affect some part of the system. If a system contains declassification, it is possible that an attacker can use the declassification mechanism to release more information than was intended. A system whose declassification is robust may release information, but it gives attackers no control over what information is released or whether information is released [85]. But, Zdancewic and Myers [127] did not propose a method for determining when a program satisfies RD. Zdancewic [125] shows that a simple change to a security type system can enforce it: extend the lattice of security labels to include integrity constraints as well as confidentiality constraints and then require that the decision to perform declassification have high integrity. Myers et al [85] introduce a type system for enforcing RD and also, they show how to support upgrading (endorsing) data integrity.

We extend our type system with a declassification instruction that perform explicit downgrading of confidential information. Also we prove that the type system is robust (in the sense of Robust Declassification introduced by Zdancewic and Myers [127]).

Rezk et al [22] provide a modular method for achieving sound type systems for declassification from sound type systems for noninterference, and instantiate this method to a sequential fragment of the JVM. They consider a declassification policy called delimited non-disclosure that combines the *what* and *where* dimensions of declassification.

Jiang, Ping and Pan [67] combined *who* and *where* dimensions to present a security model for dynamical information release and incorporated it in a new security type system which has a pair of a security level and an endorsing policy as its type. In it model, endorsing the integrity of data is controlled by the endorsing policy of each data. Such local endorsing policy describes a sequence of integrity levels through which a labeled data value may be endorsed and the conditions will hold at the execution of the corresponding endorsement points. It allows the specification of how data should be used prior to endorsement, when the endorsement is permitted, and how data should be treated after endorsement. As a result, they are able to upgrade the integrity of untrusted code in a controlled way to grant the code an ability to affect information downgrading. But, the type system enable endorsement information to a lower integrity level than the attacker level.

Myers and Askarov [13] inspired in [85] introduced a new knowledge-based framework for semantic security conditions for information security with declassification and endorsement. The impact and control of the attacker over information is characterizing in terms of sets of similar attacks. This framework, can express semantic conditions that more precisely characterize the security offered by a security type system, and derive a satisfactory account of new language features such as checked endorsement.

Part II

Justification Logic and Audited Computation

5

Justification Logic and Audited Computation

This Chapter is concerned with the computational interpretation of *Justification Logic* (formerly, the *Logic of Proofs*) [9, 10, 11] (**JL**). **JL** is a refinement of modal logic that has recently been proposed for explaining well-known paradoxes arising in the formalization of Epistemic Logic. Assertions of knowledge and belief are accompanied by *justifications*: the modality $[\![t]\!]A$ states that t is "reason" for knowing/believing A. The starting point of this work is the observation that if t is understood as a typing derivation of a term of type A, then a term of type $[\![t]\!]A$ should include some encoding of t. If this typing derivation is seen as a logical derivation, then any normalisation steps applied to it would produce a new typing derivation for A. Moreover, its relation to t would have to be made explicit in order for derivations to be closed under normalisation (in type systems parlance: for Subject Reduction (SR) to hold). This suggests that the computational interpretation of **JL** should be a programming language that records its computation history.

Structure. Section 5.3 presents an informal presentation of $\lambda^{\natural}$ by describing an example of access control based on execution history. Section 5.4 introduces $\mathbf{JL}^{\bullet}$, an affine fragment of **JL**. Sec. 5.5 studies normalisation in this system. We then introduce a term assignment for this logic in order to obtain a lambda calculus with computation history trails. This calculus is endowed with a call-by-value operational semantics and type safety of this semantics w.r.t. the type system is proved. Section 5.7 addresses strong normalisation.

5.1 Curry-Howard Isomorphism and Modal Logic

Modal logic [37] is a type of formal logic that extends the standards of formal logic to include the elements of modality. Modals qualify the truth of a judgment. Traditionally, there are three *modalities* represented in modal logic, namely, possibility, probability, and necessity. The basic modal operators are usually written $\Box$ for *necessarily* and $\Diamond$ for *Possibly*. For example, the modality $\Box A$ states that knows/believes A.

Modal formulas are defined by the following grammar:

$$F ::= P | \bot | F \supset F | \boxplus F$$

where P is a propositional variable, $\bot$ and $\supset$ are the standard propositional connectives and $\boxplus$ is one of the modalities used in a particular modal language.

The other modal and propositional connectives are defined in the standard way:

$\neg A \equiv A \supset \bot$
$A \wedge B \equiv \neg(A \supset \neg B)$
$A \leftrightarrow B \equiv (A \supset B) \wedge (B \supset A)$

$\Diamond A \equiv \neg\Box\neg A$

In order to study the logical truth and valid deductions in modal logic, we need to know how to interpret $\Box$ and $\Diamond$, i.e., we need defined precisely the meanings of *necessity* and *possibility*. There is not a unique modal logic since it is possible define infinite different modal logics. It is different from other well-known logical systems, such as classical logic and intuitionistic logic. Some of the common interpretations of $\Box A$ are:

A is necessarily true, alethic logic.
A is provable, provability logic.
It ought to be that A, deontic logic.
It is known that A, epistemic logic.
A will always be true, temporal logic.

Some common axioms and inference schemes in modal logics are:

A0. Schemes of classical propositional logic in the modal language	
A1. $\Box(F \supset G) \supset (\Box F \supset \Box G)$	*normality*
A2. $\Box F \supset F$	*reflexivity*
A3. $\Box F \supset \Box\Box F$	*Modal Positive Introspection*
A4. $\neg\Box F \supset \Box\neg\Box F$	*Modal Negative Introspection*
A5. $\Box\bot \supset \bot$	*Seriality*
R1. $F \supset G$ and F implies G	*modus ponens*
R2. $\vdash F$ implies $\vdash \Box F$	*necessitation*

The modal logic **S4** (one of the Lewis' five modal logical system) have the **A0**, **A1**, **A2**, **A3** axioms and **R1**, **R2** rules. This system is commonly studied in epistemic logic.

The Curry-Howard Isomorphism. Haskell Curry observed a correspondence between types of combinators and propositions in intuitionist implicational logic. William Howard extended this correspondence to first order logic by introducing dependent types. This correspondence is known as the Curry-Howard Isomorphism [60].

The Curry-Howard isomorphism is a correspondence between type expressions in the lambda calculus and propositional logic. The correspondence looks something like this:

Logic		**Programming**
Formulas	$\Longleftrightarrow$	Types
Proofs	$\Longleftrightarrow$	Derivations
Proof normalisation	$\Longleftrightarrow$	Reduction

The Curry-Howard Isomorphism states a correspondence between systems of formal logic and computational calculi. It has been extended to more expressive logics, e.g. higher order logic and modal logic. Through a Curry-Howard correspondence, any type system can be regarded as a logic by forgetting terms. In this sense, modal logics are contributing to practical studies for programming languages, e.g., staged computations [50], mobile code [39] and information flow analysis [78]. Modal necessity $\Box A$ may be read as the type of programs that compute values of type A and that do not depend on local resources [79, 115, 113] or resources not available at the current stage of computation [112, 123, 52]. The former reading refers to mobile computation ($\Box A$ as the type of mobile code that computes values of type A) while the latter to staged computation ($\Box A$ as the type of code that generates, at run-time, a program for computing a value of type A) [39].

Staged Computation. Staged computation refers to explicit or implicit division of a task into stages. It is a standard technique from algorithm design that has found its way to programming languages and environments. Examples are partial evaluation which refers to the global specialization of a program based on a division of the input into static (early) and dynamic (late) data, and run-time code generation which refers to the dynamic generation of optimized code based on run-time values of inputs.

David and Pfenning [50] extended the Curry-Howard correspondence to intuitionistic modal logic. They showed that the λ-calculus with the **S4** modal necessity operator $\Box$ provides a theoretical framework of staged computation by interpreting a formula $\Box A$ as a type of program codes the type A. The isomorphism relates proofs in modal logic to functional programs which manipulate program fragments for later stages. Each world in the Kripke semantics of modal logic corresponds to a stage in the computation, and a term of type $\Box A$ corresponds to code to be executed in a future stage of the computation [50].

Mobile Code. Bonelli and Feller [39] explore an intuitionistic fragment of Justification Logic (**JL**) [9, 10, 11] as a type system for a programming language for mobile units. This language caters for both code and certificate development in a unified theory. **JL** may be regarded as refinement of modal logic **S4** in which $\Box A$ is replaced by $[s]A$, for s a proof term expression, and is read: *s is a proof of A*. Bonelli and Feller interpret the modal type constructor $[s]A$ as the type of mobile units, expressions composed of a code and certificate component. In the same way that mobile code is constructed out of code components and extant type systems track local resource usage to ensure the mobile nature of these components, the Bonelli and Feller system additionally ensures correct certificate construction out of certificate components. This mechanism for internalizing its own derivations provides a natural setting for code certification.

Information Flow. Miyamoto and Igarashi [78], based-on ideas of David and Pfenning [50], develop a natural extension of the Curry-Howard isomorphism between a type system for information flow analysis and a intuitionistic modal logic. They relate the notion of security levels to possible worlds. Since information available at a lower level is also available at a higher level, a suitable modality seems to be validity $\Box_l$, which means *it is true at every level higher (or equal to) the security level l*. So, the fact that l_1 is higher than l_2 (or information can flow from l_2 to l_1) can be regarded as that a possible world l_1 is reachable from l_2 . It is expected that the proposition $\Box_l$ naturally corresponds to the type that represents the values of type A at the security level l.

5.2 JL and Hypothetical JL

In **JL** proofs are represented as combinatory terms (*proof polynomials*). Proof polynomials are constructed from proof variables and constants using two operations: application "$\cdot$" and proof-checker "!". The usual propositional connectives are augmented by a new one: given a proof polynomial s and a proposition A build $[\![s]\!]A$. The intended reading is: "*s is a proof of A*". The axioms and inference schemes of **JL** are as follows where we restrict the first axiom to minimal logic rather than classical logic:

A0. Axiom schemes of minimal logic in the language of **JL**	
A1. $[\![s]\!]A \supset A$	*"verification"*
A2. $[\![s]\!](A \supset B) \supset ([\![t]\!]A \supset [\![s \cdot t]\!]B)$	*"application"*
A3. $[\![s]\!]A \supset [\![!s]\!][\![s]\!]A$	*"proof checker"*
R1. $\Gamma \vdash A \supset B$ and $\Gamma \vdash A$ implies $\Gamma \vdash B$	*"modus ponens"*
R2. If **A** is an axiom **A0**-**A3**, and c is a proof constant, then $\vdash [\![c]\!]\mathbf{A}$	*"necessitation"*

In [12], based on work on *judgemental reconstruction* [76] of intuitionistic **S4** [50, 52, 51], judgements are introduced in which a distinction is made between propositions whose *truth* is assumed from those whose *validity* is assumed. Moreover, the notion of *proof code* is incorporated:

$$\Delta; \Gamma \vdash A \mid s$$

This judgement reads as: "*s is evidence that A is true, assuming validity of hypothesis in Δ and truth of hypothesis in Γ*". In such hypothetical judgements with proof codes, evidence s is a constituent part of

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_1}{\Delta; a : A \vdash B \mid s}}{\Delta; \cdot \vdash A \supset B \mid \lambda a^A.s}\supset\mathsf{I} \qquad \dfrac{\pi_2}{\Delta; \cdot \vdash A \mid t}}{\Delta; \cdot \vdash B \mid (\lambda a^A.s) \cdot t}\supset\mathsf{E}}{\Delta; \Gamma \vdash [\![(\lambda a^A.s) \cdot t]\!]B \mid !(\lambda a^A.s) \cdot t}\Box\mathsf{I} \qquad \dfrac{\dfrac{\pi_3}{\Delta; \cdot \vdash B \mid s^a_t}}{\Delta; \Gamma \vdash [\![(\lambda a^A.s) \cdot t]\!]B \mid !(\lambda a^A.s) \cdot t}\Box\mathsf{I}$$

Fig. 5.1. Failure of subject reduction for naive modal introduction scheme

it without which the proposed reading is no longer possible. Its importance is reflected in the following introduction rule for the $[\![s]\!]$ connective, where "$\cdot$" stands for an empty context:

$$\dfrac{\Delta; \cdot \vdash A \mid s}{\Delta; \Gamma \vdash [\![s]\!]A \mid !s}\Box\mathsf{I}$$

This scheme internalises proofs of validity: If s is evidence that A is unconditionally true, then it is true that s is a proof of A. The new witness to this fact is registered as the evidence $!s$. The "!" operator is reminiscent of that of proof polynomials. However, in our paper, proof terms such as s encode Natural Deduction and thus are no longer the proof polynomials of **JL**.

Unfortunately, the system which includes $\Box\mathsf{I}$ is not closed under substitution of derivations. For e.g. the derivation in Fig. 5.1 (left) would produce, after a step of normalisation, the derivation of Fig. 5.1 (right) where π_3 is obtained from $\pi_{1,2}$ and an appropriate substitution principle (cf. Sec. 5.4.2). However, this derivation is invalid since proof codes s^a_t (replace all free occurrences of a in s by t) and $(\lambda a^A.s) \cdot t$ do not coincide. SR may be regained, however, be introducing a judgement stating *compatibility* of proof codes, where e below is called a *compatibility code* (or trail) and is witness to the compatibility between the derivations denoted by the proof codes s and t (cf. Sec. 5.4.2 for the syntax of trails):

$$\Delta; \Gamma \vdash \mathsf{Eq}(A, s, t) \mid e \tag{5.1}$$

together with a new scheme:

$$\dfrac{\Delta; \Gamma \vdash A \mid s \quad \Delta; \Gamma \vdash \mathsf{Eq}(A, s, t) \mid e}{\Delta; \Gamma \vdash A \mid t}\mathsf{Eq}$$

Normalisation of derivations gives rise to instances of Eq. These instances in fact may be permuted past any inference scheme *except* a box introduction. This suggests a normalisation procedure over *canonical* derivations where instances of Eq are permuted until they reach the innermost instance of a $\Box$ introduction scheme at which point they permute to the upper right-hand hypothesis (cf. Sec. 5.5):

$$\dfrac{\Delta; \cdot \vdash A \mid s \quad \Delta; \cdot \vdash \mathsf{Eq}(A, s, t) \mid e}{\Delta; \Gamma \vdash [\![t]\!]A \mid t}\Box\mathsf{I}$$

It determines a natural notion of *locality* of trails given in terms of *scope*: trails are local to the innermost box introduction scheme. The resulting calculus, may be shown strongly normalising and confluent by applying properties of higher-order rewrite systems [12].

Although reduction and their corresponding trails are confined to local computation units, computation itself is unaware of them. We thus incorporate means for *reification* of trails. We introduce *trail variables* in order to name the trail of a local computation and extend the introduction rule for the box accordingly:

$$\dfrac{\Delta; \cdot; \Sigma \vdash A \mid s \quad \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e}{\Delta; \Gamma; \Sigma' \vdash [\![\Sigma.t]\!]A \mid \Sigma.t}\Box\mathsf{I}$$

Here Σ is a context of *affine* trail variables. Trail variables are affine since each trail lookup may produce a different result. The proof code $\Sigma.t$ binds all free occurrences of the trail variables in Σ. A *fold* over trails is also introduced (cf. TI in Sec. 5.4).

The resulting presentation of **JL** we coin the *hypothetical* **JL** and denote it with $\mathbf{JL}^\bullet$. Its schemes are now described in further detail from the point of view of $\lambda^\mathfrak{h}$, the induced *calculus of audited units* via its term assignment.

5.3 A Calculus of Audited Units

We begin with an informal presentation of $\lambda^\mathfrak{h}$ by describing an example program that models Abadi and Fournet's [5] mechanism for access control based on execution history. This mechanism, proposed as an enhanced alternative to Java stack-inspection, consists in controlling access to objects depending on which functions were invoked previously in time. Each function is (statically) assigned a set of permissions; the current set of permissions is computed by taking the intersection of the permissions of all function calls in the history of the computation. As an example, consider a function **deleteFile** that given a file name, either deletes the file if it has the appropriate permission (*FileIOPerm*) or raises an exception if it does not. In $\lambda^\mathfrak{h}$ we write the top-level declaration:

$$\mathbf{deleteFile} \doteq !^{\alpha}\lambda a^{Str}.\mathsf{if}\ \mathit{FileIOPerm} \in \alpha\vartheta\ \mathsf{then}\ \mathbf{Win32Delete}\, a\ \mathsf{else}\ \mathbf{securityException}; \tag{5.2}$$

An expression of the form $!_e^{\alpha_1,\dots,\alpha_n}M$ is called an *audited (computation) unit*, M being the *body*, e the *history* or *trail* of computation producing M and α_i, $i \in 1..n$, the *trail variables* that are used for consulting the computation history. In this example, there is only one trail variable, namely α; its scope is the innermost enclosing *modal term constructor* "$!^\alpha$". And, $\alpha\vartheta$ is a occurrence of trail variable, where α is a trail variable and ϑ is the evaluation of trail (trail inspection). As discussed, the trail e is an encoding of all reduction steps that have taken place from some origin term to produce the current term M. Uninteresting trails, such as those witnessing that no computation has taken place (e.g. that the type derivation for M is equal to itself), are often omitted for notational convenience, as has been done in (5.2). Next we describe some of the salient features of $\lambda^\mathfrak{h}$, namely (1) trail update; (2) substitution of audited units; and (3) trail inspection.

Trail update. Consider the term $!^{\alpha}_{\mathfrak{r}((\lambda a^{Str}.s_1)\cdot s_2)}M$, where M is:

$$\left(\lambda a^{Str}.\mathsf{if}\ \mathit{FileIOPerm} \in \vartheta\alpha\ \mathsf{then}\, \mathbf{Win32Delete}\, a\ \mathsf{else}\, \mathbf{securityException}\right)\ \text{“..}\backslash \mathit{passwd}\text{''}$$

It differs in two ways w.r.t. (5.2). First the body M of the audited unit is an application rather than an abstraction. Second, the uninteresting trail reflecting that no computation has taken place is now displayed: $\mathfrak{r}((\lambda a^{Str}.s_1)\cdot s_2)$. This trail asserts that $(\lambda a^{Str}.s_1)\cdot s_2$ is the code of a typing derivation for M and should be considered compatible with itself. It reflects that no computation has taken place since it does not have occurrences of β-trails, as we now describe. A β-step of computation from $!^{\alpha}_{\mathfrak{r}((\lambda a^{Str}.s_1)\cdot s_2)}M$ produces:

$$\begin{array}{l} !^{\alpha}_{\mathfrak{t}(\mathfrak{ba}(a^{Str}.s_1,s_2),\mathfrak{r}((\lambda a^{Str}.s_1)\cdot s_2))}\ \mathsf{if}\ \mathit{FileIOPerm} \in \alpha\vartheta \\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{then}\, \mathbf{Win32Delete}\ \text{“..}\backslash\mathit{passwd}\text{''} \\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\, \mathbf{securityException} \end{array} \tag{5.3}$$

The trail of the audited unit has been updated to reflect the reduction step that took place. In $\mathfrak{t}(\mathfrak{ba}(a^{Str}.s_1, s_2), \mathfrak{r}((\lambda a^{Str}.s_1)\cdot s_2))$, the expressions $a^{Str}.s_1, s_2$ and $(\lambda a^{Str}.s_1)\cdot s_2$ are encodings of typing derivations for $(\lambda a^{Str}.\mathsf{if}\ \mathit{FileIOPerm} \in \alpha\vartheta\, \mathsf{then}\, \mathbf{Win32Delete}\, a\, \mathsf{else}\, \mathbf{securityException})$, "..$\backslash$*passwd*" and M, resp. The β-trail $\mathfrak{ba}(a^{Str}.s_1, s_2)$ states that the typing derivation s_1 where all free occurrences of a have been replaced by s_2 should be considered compatible with $(\lambda a^{Str}.s_1)\cdot s_2$; the trail constructor $\mathfrak{t}$ asserts transitivity of compatibility.

This account of the process of updating the trail while computing is slightly abridged; it is actually developed in two stages: a *principle contraction*, which produces the trail, followed by a number of *permutative conversions*, which transports the trail to its innermost enclosing audited unit constructor (cf. Sec. 5.5).

Substitution for audited units. Let us return to **deleteFile**. How do we supply the name of the file to delete? Clearly we cannot simply apply **deleteFile** to a string since it is not an abstraction (indeed, it is an audited unit). We require some means of extracting the value computed by an audited unit. This is accomplished by *audited unit composition* $\mathsf{let}\, u = M \,\mathsf{in}\, N$. We refer to M as its *argument* and N its *body*. It evaluates as follows, determining a $\beta_\Box$-step. First evaluate M until a value $!_e^{\alpha_1,\ldots,\alpha_n} V$ is obtained (note the V under the modal term constructor). Then replace the free occurrences of u in N with V. In addition, the following actions are performed, as dictated by the proof theoretical analysis that shall be developed shortly:

1. e is copied so that trails are correctly persisted; and
2. all free occurrences of $\alpha_1, \ldots, \alpha_n$ in V are "rewired" with the trail variables of the (innermost) audited unit in which u resides.[1] Trail variables u are therefore accompanied by a trail variable rewiring and are written $\langle u; \{\alpha_1/\beta_1, \ldots, \alpha_n/\beta_n\}\rangle$.

As an illustration, consider the following additional top-level declarations:

$$\begin{array}{ll}\mathbf{cleanup} & \doteq\, !^{\beta} \lambda a.\mathbf{delete}\ \beta\ a; \\ \mathbf{bad} & \doteq\, !^{\gamma} \mathbf{cleanup}\ \gamma\ \text{“..}\backslash passwd\text{''};\end{array}$$

where we write $\mathbf{f}\,\vec{\beta}\,\vec{N}$ to abbreviate $\mathsf{let}\, u = \mathbf{f} \,\mathsf{in}\, \langle u; \vec{\alpha}/\vec{\beta}\rangle\, \vec{N}$ assuming that $\mathbf{f} \doteq !_e^{\vec{\alpha}} \lambda\vec{a} : \vec{A}.M$. Evaluation of the term $!^{\delta}\mathbf{bad}\ \delta$ will produce:

$$!_e^{\delta} \mathsf{if}\ \mathit{FileIOPerm} \in \delta\vartheta\ \mathsf{then}\ \mathbf{Win32Delete}\ \text{“..}\backslash passwd\text{''}\ \mathsf{else}\ \mathbf{securityException} \tag{5.4}$$

The trail e will include three instances of a $\mathfrak{bb}$ trail constructor reflecting three $\beta_\Box$-steps and the $\mathfrak{def}_{\mathbf{deleteFile}}$, $\mathfrak{def}_{\mathbf{cleanup}}$ and $\mathfrak{def}_{\mathbf{bad}}$ trail constructors reflecting three top-level function unfoldings. The next step is to determine whether the predicate $\mathit{FileIOPerm} \in \delta\vartheta$ holds. This brings us to the last part of our informal introduction to $\lambda^{\mathfrak{h}}$, namely *trail inspection*, after which we shall complete the description of our example.

Trail inspection. Inspection of trails is achieved by means of trail variables. Evaluation of trail variables inside an audited unit consists in first looking up the trail and then immediately traversing it, replacing each constructor of the trail with a term of the appropriate type[2]. The mapping that replaces trail constructors with terms is called a *trail replacement*. All occurrences of trail variables are thus written $\alpha\vartheta$ where α is a trail variable and ϑ a trail replacement. A notable property of trail variables is that they are *affine* (i.e. at most one permitted use) since each trail inspection may produce a different result. Returning to (5.4), assume we have at our disposal an assignment of a set of (static) permissions to top-level functions: $\mathit{perms}(\mathbf{bad}) \overset{\text{def}}{=} \emptyset$, $\mathit{perms}(\mathbf{cleanup}) \overset{\text{def}}{=} \{\mathit{FileIOPerm}\}$ and $\mathit{perms}(\mathbf{deleteFile}) \overset{\text{def}}{=} \{\mathit{FileIOPerm}\}$. Assume, moreover, that the trail replacement takes the form:

$$\begin{array}{rlrl}\vartheta(\mathfrak{r}) = \vartheta(\mathfrak{ti}) & \overset{\text{def}}{=} \emptyset & \vartheta(\mathfrak{rw}) & \overset{\text{def}}{=} \lambda\overline{a}^{\mathbb{N}}.a_1 \cap .. \cap a_{10} \\ \vartheta(\mathfrak{s}) = \vartheta(\mathfrak{ab}) & \overset{\text{def}}{=} \lambda a^{\mathbb{N}}.a & \vartheta(\mathfrak{b}) = \vartheta(\mathfrak{bb}) & \overset{\text{def}}{=} \emptyset \\ \vartheta(\mathfrak{t}) = \vartheta(\mathfrak{ap}) = \vartheta(\mathfrak{l}) & \overset{\text{def}}{=} \lambda a^{\mathbb{N}}.\lambda b^{\mathbb{N}}.a \cap b & \vartheta(\mathfrak{def}_{\mathbf{f}}) & \overset{\text{def}}{=} \{\mathit{perms}(\mathbf{f})\}\end{array}$$

Returning to (5.4), evaluation of $\delta\vartheta$ produces the empty set given that the intersection of the permissions associated to the trail constructors for $\mathfrak{def}_{\mathbf{deleteFile}}$, $\mathfrak{def}_{\mathbf{cleanup}}$ and $\mathfrak{def}_{\mathbf{bad}}$ is the empty set. Thus the predicate $\mathit{FileIOPerm} \in \delta\vartheta$ is false and a security exception is raised.

[1] These two items illustrate how the reduction behavior of $\mathsf{let}\, u = M \,\mathsf{in}\, N$ differs from the standard computational interpretation of some modal logics [91, 52, 51].

[2] In the same way as one recurs over lists using fold in functional programming, replacing nil and cons by appropriate terms.

5.3.1 History based access control for information flow

The next example is drawn from studies in static checking of method bodies in object-oriented languages to ensure that they satisfy confidentiality policies. These policies guarantee that private information is not leaked into public channels. They should be flexible enough to admit the largest possible number of programs. With that objective in mind, it would be convenient to allow a method to be given several types so that different information flow policies can be imposed for callers with different permissions. We explain the idea with the following example from [18]. Consider a trusted function **getStatus** that can be called in more than one context. If called by untrusted code, **getStatus** returns public information. Trusted code, however, can obtain private information.

The definition of *perms* is as follows: $perms(\mathbf{get}) \stackrel{\text{def}}{=} \emptyset$, $perms(\mathbf{getStatus}) \stackrel{\text{def}}{=} \emptyset$ and $perms(\mathbf{getHinfo}) \stackrel{\text{def}}{=} \{sysPerm\}$. The trail replacement ϑ is the same as in the previous example. The top-level declarations we have at our disposal are:

$$\begin{array}{ll}
\mathbf{getHinfo} & \doteq\, !^{\alpha_{gH}}_{Refl(r)} \mathsf{if}\ sysPerm \in \theta\alpha_{gH} \\
& \qquad\qquad \mathsf{then}\ \mathbf{privateInfo} \\
& \qquad\qquad \mathsf{else}\ \mathbf{securityException}; \\
\mathbf{getStatus} & \doteq\, !^{\alpha_{gS}}_{Refl(s)} \mathsf{if}\ sysPerm \in \theta\alpha_{gS} \\
& \qquad\qquad \mathsf{then}\ \mathbf{privateInfo} \\
& \qquad\qquad \mathsf{else}\ \mathbf{publicInfo};
\end{array}$$

Function **getHinfo** is useful only to callers with permission *sysPerm*. However, **getStatus** is useful both for callers with permission *sysPerm* and for those without; only the former obtain private info. Volpano-Irvine-Smith style type analysis [118] would assign **getStatus** the type H. The type system of [18] introduces permissions into types and assigns the more precise type[3] $\overline{\{sysPerms\}} \to L$ reflecting that the result may be public (indicated by the "L") if *sysPerms* is absent.

5.4 The Logic

JL is a modal logic of provability which has a sound and complete arithmetical semantics. This section introduces a natural deduction presentation for a fragment[4] of **JL**. The inference schemes we shall define give meaning to *hypothetical judgements with proof codes* $\Delta; \Gamma; \Sigma \vdash A \mid s$ whose intended reading is: "*s is evidence that A is true under validity hypothesis Δ, truth hypothesis Γ and compatibility hypothesis Σ*".

We assume given term variables $a, b, \ldots$, audited unit variables $u, v, \ldots$ and trail variables $\alpha, \beta, \ldots$. The syntax of each component of the judgement is described below:

$$\begin{array}{rrl}
\text{Propositions} & A & ::= P \mid A \supset A \mid [\![\Sigma.s]\!]A \\
\text{Validity context} & \Delta & ::= \cdot \mid \Delta, u : A[\Sigma] \\
\text{Truth context} & \Gamma & ::= \cdot \mid \Gamma, a : A \\
\text{Compat. context} & \Sigma & ::= \cdot \mid \Sigma, \alpha : \mathsf{Eq}(A) \\
\text{Rewiring} & \sigma & ::= \{\alpha_1/\beta_1, \ldots, \alpha_n/\beta_n\} \\
\text{Proof code} & s & ::= a \mid \lambda a^A.s \mid s \cdot s \mid \langle u; \sigma \rangle \mid \Sigma.s \mid \text{LET}(u^{A[\Sigma]}.s, s) \mid \alpha\theta \\
\text{Proof code trail replmnt} & \theta & ::= \{\mathfrak{r}/s_1, \mathfrak{s}/s_2, \mathfrak{t}/s_3, \mathfrak{ba}/s_4, \mathfrak{bb}/s_5, \mathfrak{ti}/s_6, \mathfrak{ab}\mathfrak{C}/s_7, \mathfrak{ap}\mathfrak{C}/s_8, \mathfrak{le}\mathfrak{C}/s_9, \mathfrak{rp}\mathfrak{C}/s_{10}\}
\end{array}$$

Contexts are considered multisets; "$\cdot$" denotes the empty context. In $\Delta; \Gamma; \Sigma$ we assume all variables to be fresh. Variables in Σ are assigned a type of the form $\mathsf{Eq}(A)$[5]. A proposition is either a propositional

[3] Actually, this is a simplification of the type assigned in [18] since there they study an object-oriented object language.

[4] Minimal propositional **JL** without the "plus" polynomial proof constructor.

[5] The type $\mathsf{Eq}(A)$ in an assignment $\alpha : \mathsf{Eq}(A)$ may informally be understood as $\exists x, y.\mathsf{Eq}(A, x, y)$ (where x, y stand for arbitrary type derivations of propositions of type A) since α stands for a proof of compatibility of two type derivations of propositions of type A about which nothing more may be assumed. These type derivations are hidden since trail inspection may take place at any time.

$$\frac{a : A \in \Gamma}{\Delta; \Gamma; \Sigma \vdash A \mid a}\,\mathsf{Var} \qquad \frac{\Delta; \Gamma, a : A; \Sigma \vdash B \mid s}{\Delta; \Gamma; \Sigma \vdash A \supset B \mid \lambda a^{A}.s} \supset \mathsf{I}$$

$$\frac{\begin{array}{c}\Delta; \Gamma_1; \Sigma_1 \vdash A \supset B \mid s \\ \Delta; \Gamma_2; \Sigma_2 \vdash A \mid t\end{array}}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash B \mid s \cdot t} \supset \mathsf{E} \qquad \frac{u : A[\Sigma] \in \Delta \quad \Sigma\sigma \subseteq \Sigma'}{\Delta; \Gamma; \Sigma' \vdash A \mid \langle u; \sigma \rangle}\,\mathsf{mVar}$$

$$\frac{\begin{array}{c}\Delta; \cdot; \Sigma \vdash A \mid s \\ \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e\end{array}}{\Delta; \Gamma; \Sigma' \vdash [\![\Sigma.t]\!]A \mid \Sigma.t}\,\Box\mathsf{I} \qquad \frac{\begin{array}{c}\Delta; \Gamma_1; \Sigma_1 \vdash [\![\Sigma.r]\!]A \mid s \\ \Delta, u : A[\Sigma]; \Gamma_2; \Sigma_2 \vdash C \mid t\end{array}}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash C^{u}_{\Sigma.r} \mid \textsc{let}(u^{A[\Sigma]}.t, s)}\,\Box\mathsf{E}$$

$$\frac{\alpha : \mathsf{Eq}(A) \in \Sigma \quad \Delta; \cdot; \cdot \vdash \mathcal{T}^{B} \mid \theta}{\Delta; \Gamma; \Sigma \vdash B \mid \alpha\theta}\,\mathsf{TI} \qquad \frac{\Delta; \Gamma; \Sigma \vdash A \mid s \quad \Delta; \Gamma; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e}{\Delta; \Gamma; \Sigma \vdash A \mid t}\,\mathsf{Eq}$$

Fig. 5.2. Inference Schemes for Hypothetical Judgements with Proof Codes

variable P, an implication $A \supset B$ or a modality $[\![\Sigma.s]\!]A$. In $[\![\Sigma.s]\!]A$, "Σ." binds all occurrences of trail variables in s and hence may be renamed at will. We refer to an encoding of a type derivation as *proof code*. Proof code bears witness to proofs of propositions, they encode each possible scheme that may be applied: truth hypothesis, abstraction, audited computation unit variable, audited computation unit introduction and elimination, and trail inspection. A compatibility code replacement is a mapping θ from the set of trail constructors (introduced shortly) to proof codes. It is used for associating proof codes to trail replacements. We write σ for for a bijective map over trail variables that we call rewiring. Free truth variables of s ($\mathsf{fvT(s)}$), free validity variables of s ($\mathsf{fvV(s)}$) and free trail variables of s ($\mathsf{fvTrl(s)}$) are defined as expected.

5.4.1 Inference Schemes

The meaning of hypothetical judgements with proof codes is given by the axiom and inference schemes of Fig. 5.2 and determine the $\mathbf{JL}^{\bullet}$ *system*. Both Γ and Σ are affine hypothesis whereas those in Δ are intuitionistic. We briefly comment on the schemes. The axiom scheme Var states that judgement "$\Delta; \Gamma; \Sigma \vdash A \mid a$" is evident in itself: if we assume a is evidence that proposition A is true, then we may immediately conclude that A is true with proof code a. Similarly, the asumption that A is valid allows us to conclude that it is true, as indicated by mVar. However, since the validity of A depends on compatiblity assumptions Σ, they must be "rewired" to fit the current context. The schemes for introduction and elimination of implication need no further explanation. The introduction scheme for the modality has already been motivated in the introduction. The compatibility code e of $\Box\mathsf{I}$ can take one of the following forms, where $\mathfrak{rpC}(e_1, \dots, e_{10})$ is usually abbreviated $\mathfrak{rpC}(\overline{e})$:

$e ::= \mathfrak{r}(s)$	reflexivity
$\mid\ \mathfrak{s}(e)$	symmetry
$\mid\ \mathfrak{t}(e, e)$	transitivity
$\mid\ \mathfrak{ba}(a^{A}.s, s)$	β
$\mid\ \mathfrak{bb}(u^{A[\Sigma]}.s, \Sigma.s)$	$\beta_{\Box}$
$\mid\ \mathfrak{ti}(\theta, \alpha)$	trail inspection
$\mid\ \mathfrak{abC}(a^{A}.e)$	abstraction compatibility
$\mid\ \mathfrak{apC}(e, e)$	application compatibility
$\mid\ \mathfrak{leC}(u^{A[\Sigma]}.e, e)$	let compatibility
$\mid\ \mathfrak{rpC}(e_1, \dots, e_{10})$	replacement compatibility

$$\frac{\Delta;\Gamma;\Sigma \vdash A \mid s}{\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A,s,s) \mid \mathfrak{r}(s)}\ \mathsf{EqRefl} \qquad \frac{\begin{array}{c}\Delta;\Gamma_1, a:A;\Sigma_1 \vdash B \mid s\\ \Delta;\Gamma_2;\Sigma_2 \vdash A \mid t\end{array}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \mathsf{Eq}(B, s^a_t, (\lambda a^A.s)\cdot t) \mid \mathfrak{ba}(a^A.s,t)}\ \mathsf{Eq}\beta$$

$$\frac{\Delta;\cdot;\Sigma_1 \vdash A \mid r \quad \Delta;\cdot;\Sigma_1 \vdash \mathsf{Eq}(A,r,s) \mid e \quad \Delta, u:A[\Sigma_1];\Gamma_2;\Sigma_2 \vdash C \mid t \quad \Gamma_2 \subseteq \Gamma_3 \quad \Sigma_2 \subseteq \Sigma_3}{\Delta;\Gamma_3;\Sigma_3 \vdash \mathsf{Eq}(C^u_{\Sigma_1.s}, t^u_{\Sigma_1.s}, \textsc{let}(u^{A[\Sigma_1]}.t, \Sigma_1.s)) \mid \mathfrak{bb}(u^{A[\Sigma_1]}.t, \Sigma_1.s)}\ \mathsf{Eq}\beta_\Box$$

$$\frac{\Delta;\cdot;\Sigma_1 \vdash \mathsf{Eq}(A,s,t) \mid e \quad \Delta;\cdot;\cdot \vdash \mathcal{T}^B \mid \theta \quad \alpha : \mathsf{Eq}(A) \in \Sigma_2}{\Delta;\Gamma;\Sigma_2 \vdash \mathsf{Eq}(B, e\theta, \alpha\theta) \mid \mathfrak{ti}(\theta,\alpha)}\ \textsc{EqTI}$$

$$\frac{\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A,s,t) \mid e}{\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A,t,s) \mid \mathfrak{s}(e)}\ \mathsf{EqSym} \qquad \frac{\begin{array}{c}\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A,s_1,s_2) \mid e_1\\ \Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A,s_2,s_3) \mid e_2\end{array}}{\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A,s_1,s_3) \mid \mathfrak{t}(e_1,e_2)}\ \mathsf{EqTrans}$$

$$\frac{\Delta;\Gamma, a:A;\Sigma \vdash \mathsf{Eq}(B,s,t) \mid e}{\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A \supset B, \lambda a^A.s, \lambda a^A.t) \mid \mathfrak{abC}(a^A.e)}\ \mathsf{EqAbs} \qquad \frac{\begin{array}{c}\Delta;\Gamma_1;\Sigma_1 \vdash \mathsf{Eq}(A \supset B, s_1, s_2) \mid e_1\\ \Delta;\Gamma_2;\Sigma_2 \vdash \mathsf{Eq}(A,t_1,t_2) \mid e_2\end{array}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \mathsf{Eq}(B, s_1\cdot t_1, s_2 \cdot t_2) \mid \mathfrak{apC}(e_1,e_2)}\ \mathsf{EqApp}$$

$$\frac{\Delta;\Gamma_1;\Sigma_1 \vdash \mathsf{Eq}([\![\Sigma.r]\!]A, s_1, s_2) \mid e_1 \quad \Delta, u:A[\Sigma];\Gamma_2;\Sigma_2 \vdash \mathsf{Eq}(C,t_1,t_2) \mid e_2}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \mathsf{Eq}(C^u_{\Sigma.r}, \textsc{let}(u^{A[\Sigma]}.t_1, s_1), \textsc{let}(u^{A[\Sigma]}.t_2, s_2)) \mid \mathfrak{leC}(u^{A[\Sigma]}.e_2, e_1)}\ \mathsf{EqLet}$$

$$\frac{\Delta;\cdot;\cdot \vdash \mathsf{Eq}(\mathcal{T}^B, \theta', \theta) \mid e_i \quad \alpha : \mathsf{Eq}(A) \in \Sigma}{\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(B, \alpha\theta', \alpha\theta) \mid \mathfrak{rpC}(\overline{e})}\ \textsc{EqRpl}$$

Fig. 5.3. Schemes defining proof code compatibility judgement

The schemes defining the judgement (5.1) are given in Fig. 5.3. There are four proof code compatibility axioms (EqRefl, $\mathsf{Eq}\beta$, $\mathsf{Eq}\beta_\Box$ and $\textsc{EqTI}$) and six inference schemes (the rest). The axioms are used for recording principle contractions (Sec. 5.5) at the root of a term and schemes EqAbs, EqApp, EqLet and $\textsc{EqTI}$ enable the same recording but under each of the term constructors. Note that there are no congruence schemes for the modality. Regarding trail inspection ($\textsc{EqTI}$ in Fig. 5.3) recall from the introduction that we append each reference to a trail variable with a trail replacement. Therefore, the trail replacement has to bee accompanied by proof codes, one for each term that is to replace a trail constructor. The proof code for each of these proofs is grouped as θ and is called a *proof code trail replacement*: $\Delta;\cdot;\cdot \vdash \mathcal{T}^B \mid \theta$ which is a shorthand for $\Delta;\cdot;\cdot \vdash \mathcal{T}^B(c) \mid \theta(c)$, for each c in the set of compatibility witness constructors $\{\mathfrak{r}, \mathfrak{s}, \mathfrak{t}, \mathfrak{ba}, \mathfrak{bb}, \mathfrak{ti}, \mathfrak{abc}, \mathfrak{apc}, \mathfrak{lec}, \mathfrak{rpc}\}$, where $\mathcal{T}^B(c)$ is the type of term that replaces the trail constructor c. These types are defined as follows:

$$\begin{array}{rclrcl}
\mathcal{T}^B(\mathfrak{r}) & \overset{\text{def}}{=} & B & \mathcal{T}^B(\mathfrak{abc}) & \overset{\text{def}}{=} & B \supset B\\
\mathcal{T}^B(\mathfrak{s}) & \overset{\text{def}}{=} & B \supset B & \mathcal{T}^B(\mathfrak{apc}) & \overset{\text{def}}{=} & B \supset B \supset B\\
\mathcal{T}^B(\mathfrak{t}) & \overset{\text{def}}{=} & B \supset B \supset B & \mathcal{T}^B(\mathfrak{lec}) & \overset{\text{def}}{=} & B \supset B \supset B\\
\mathcal{T}^B(\mathfrak{ba}) = \mathcal{T}^B(\mathfrak{bb}) & \overset{\text{def}}{=} & B & \mathcal{T}^B(\mathfrak{rpc}) & \overset{\text{def}}{=} & \underbrace{B \supset \ldots \supset B}_{10\text{ copies}} \supset B\\
\mathcal{T}^B(\mathfrak{ti}) & \overset{\text{def}}{=} & B & & &
\end{array}$$

Remark 1 It should be noted that the proof code s in a derivable judgement $\Delta;\Gamma;\Sigma \vdash A \mid s$ does not encode a derivation of this judgement due to the presence of Eq. The term assignment for $\mathbf{JL}^\bullet$ is the topic of Sec. 5.6.

5.4.2 Basic Metatheoretic Results

Some basic meta-theoretic results about $\mathbf{JL}^\bullet$ are presented next. The judgements in the statement of these results are decorated with terms (such as M, N and $M^a_{N,t}$ below) which may safely be ignored for the time being (they are introduced in Sec. 5.6 when we discuss the term assignment for $\mathbf{JL}^\bullet$). The first states that hypothesis may be added without affecting derivability.

Lemma 5.4.1 (Weakening) 1. If $\Delta;\Gamma;\Sigma \vdash M : A \mid s$ is derivable, then so is $\Delta';\Gamma';\Sigma' \vdash M : A \mid s$, where $\Delta \subseteq \Delta'$, $\Gamma \subseteq \Gamma'$ and $\Sigma \subseteq \Sigma'$.
2. If $\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A,s,t) \mid e$ is derivable, then so is $\Delta';\Gamma';\Sigma' \vdash \mathsf{Eq}(A,s,t) \mid e$, where $\Delta \subseteq \Delta'$, $\Gamma \subseteq \Gamma'$ and $\Sigma \subseteq \Sigma'$.

The second and third results address two substitution principles. The first principle is about substitution of truth variables. We abbreviate Γ_1,Γ_2 with $\Gamma_{1,2}$. If $\Gamma = \Gamma_1, a : A, \Gamma_3$, we write $\Gamma^a_{\Gamma_2}$ for $\Gamma_{1,2,3}$. Also, we write s^a_t for the substitution of all free occurrences of a in s by t and define it as follows (and similarly for e^a_s), where in the third clause we assume that b has been renamed away from s:

$$
\begin{array}{rcl@{\qquad}rcl}
 & & & \mathfrak{r}(t)^a_s & \overset{\text{def}}{=} & \mathfrak{r}(t^a_s)\\
a^a_s & \overset{\text{def}}{=} & s & \mathfrak{s}(e)^a_s & \overset{\text{def}}{=} & \mathfrak{s}(e^a_s)\\
b^a_s & \overset{\text{def}}{=} & b & \mathfrak{t}(e_1,e_2)^a_s & \overset{\text{def}}{=} & \mathfrak{t}({e_1}^a_s,{e_2}^a_s)\\
(\lambda b^A.t)^a_s & \overset{\text{def}}{=} & \lambda b^A.t^a_s & \mathfrak{ba}(b^A.r,t)^a_s & \overset{\text{def}}{=} & \mathfrak{ba}(b^A.r^a_s,t^a_s)\\
(t_1 \cdot t_2)^a_s & \overset{\text{def}}{=} & {t_1}^a_s \cdot {t_2}^a_s & \mathfrak{bb}(u^{A[\Sigma]}.r,\Sigma.t)^a_s & \overset{\text{def}}{=} & \mathfrak{bb}(u^{A[\Sigma]}.r^a_s,\Sigma.t)\\
\langle u;\sigma\rangle^a_s & \overset{\text{def}}{=} & \langle u;\sigma\rangle & \mathfrak{ti}(\theta,\alpha)^a_s & \overset{\text{def}}{=} & \mathfrak{ti}(\theta,\alpha)\\
(\Sigma.t)^a_s & \overset{\text{def}}{=} & \Sigma.t & \mathfrak{abC}(b^A.e)^a_s & \overset{\text{def}}{=} & \mathfrak{abC}(b^A.e^a_s)\\
\text{LET}(u^{A[\Sigma]}.t_2,t_1)^a_s & \overset{\text{def}}{=} & \text{LET}(u^{A[\Sigma]}.{t_2}^a_s,{t_1}^a_s) & \mathfrak{apC}(e_1,e_2)^a_s & \overset{\text{def}}{=} & \mathfrak{apC}({e_1}^a_s,{e_2}^a_s)\\
(\alpha\theta)^a_s & \overset{\text{def}}{=} & \alpha\theta & \mathfrak{leC}(u^{A[\Sigma]}.e_1,e_2)^a_s & \overset{\text{def}}{=} & \mathfrak{leC}(u^{A[\Sigma]}.{e_1}^a_s,{e_2}^a_s)\\
 & & & \mathfrak{rpC}(\overline{e})^a_s & \overset{\text{def}}{=} & \mathfrak{rpC}(\overline{e^a_s})
\end{array}
$$

Lemma 5.4.2 (Subst. Principle for Truth Hypothesis) Suppose $\Delta;\Gamma_2;\Sigma_2 \vdash N : A \mid t$ is derivable and $a : A \in \Gamma_1$.

1. If $\Delta;\Gamma_1;\Sigma_1 \vdash M : B \mid s$, then $\Delta;{\Gamma_1}^a_{\Gamma_2};\Sigma_{1,2} \vdash M^a_{N,t} : B \mid s^a_t$.
2. If $\Delta;\Gamma_1;\Sigma_1 \vdash \mathsf{Eq}(B,s_1,s_2) \mid e$, then $\Delta;{\Gamma_1}^a_{\Gamma_2};\Sigma_{1,2} \vdash \mathsf{Eq}(B,(s_1)^a_t,(s_2)^a_t) \mid e^a_t$.

The second substitution principle is about substitution of validity variables. Substitution of validity variables in proof codes is denoted $s^u_{\Sigma.t}$. It is defined as follows, where C^u_s stands for the proposition resulting from replacing all occurrences of u in C with s; in the second clause the domain of σ is Σ; and all trail variables in Σ are rewired by means of the rewiring σ:

$$
\begin{array}{rcl}
a^u_{\Sigma.s} & \overset{\text{def}}{=} & a\\
\langle u;\sigma\rangle^u_{\Sigma.s} & \overset{\text{def}}{=} & s\sigma\\
\langle v;\sigma\rangle^u_{\Sigma.s} & \overset{\text{def}}{=} & \langle v;\sigma\rangle\\
(\lambda b^A.t)^u_{\Sigma.s} & \overset{\text{def}}{=} & \lambda b^A.t^u_{\Sigma.s}\\
(t_1 \cdot t_2)^u_{\Sigma.s} & \overset{\text{def}}{=} & {t_1}^u_{\Sigma.s} \cdot {t_2}^u_{\Sigma.s}\\
(\Sigma'.t)^u_{\Sigma.s} & \overset{\text{def}}{=} & \Sigma'.t^u_{\Sigma.s}\\
\text{LET}(v^{A[\Sigma']}.t_2,t_1)^u_{\Sigma.s} & \overset{\text{def}}{=} & \text{LET}(v^{A[\Sigma']}.{t_2}^u_{\Sigma.s},{t_1}^u_{\Sigma.s})\\
(\alpha\theta)^u_{\Sigma.s} & \overset{\text{def}}{=} & \alpha(\theta)^u_{\Sigma.s}
\end{array}
$$

Lemma 5.4.3 (Subst. Principle for Validity Hypothesis) Suppose judgements $\Delta_{1,2};\cdot;\Sigma_1 \vdash M : A \mid s$ and $\Delta_{1,2};\cdot;\Sigma_1 \vdash \mathsf{Eq}(A,s,t) \mid e_1$ are derivable. Let $\Delta \overset{\text{def}}{=} \Delta_1, u : A[\Sigma_1], \Delta_2$. Then:

1. If $\Delta;\Gamma;\Sigma_2 \vdash N : C \mid r$, then $\Delta_{1,2};\Gamma;\Sigma_2 \vdash N^u_{\Sigma_1.(M,t,e_1)} : C^u_{\Sigma_1.t} \mid r^u_{\Sigma_1.t}$.

2. If $\Delta;\Gamma;\Sigma_2 \vdash \mathsf{Eq}(C,s_1,s_2) \mid e_2$, then $\Delta_{1,2};\Gamma;\Sigma_2 \vdash \mathsf{Eq}(C^u_{\Sigma_1.t}, s_1{}^u_{\Sigma_1.t}, s_2{}^u_{\Sigma_1.t}) \mid e_2{}^u_{\Sigma_1.t}$.

Remark 2 In this substitution principle, substitution of $u : A[\Sigma_1]$ requires not only a derivation of $\Delta_{1,2};\cdot;\Sigma_1 \vdash M : A \mid s$, but *also* its computation history $\Delta_{1,2};\cdot;\Sigma_1 \vdash \mathsf{Eq}(A,s,t) \mid e_1$ (cf. substitution of validity variables, in particular the clause for $\langle u;\sigma\rangle$, in Sec. 5.6)

The last ingredient we require before discussing normalisation is the following lemma which is used for computing the results of trail inspection. If ϑ is a mapping from trail constructors to terms, then $e\vartheta$ produces a *term* by replacing each trail constructor in e by its associated term via ϑ. For example, $\mathfrak{ba}(a^A.r,t)\vartheta \overset{\text{def}}{=} \vartheta(\mathfrak{ba})$ and $\mathfrak{t}(e_1,e_2)\vartheta \overset{\text{def}}{=} \vartheta(\mathfrak{t})\, e_1\vartheta\, e_2\vartheta$. In contrast, $e\theta$ produces a *proof code* by replacing each trail constructor in e with its associated proof code via θ.

Lemma 5.4.4 $\Delta;\cdot;\cdot \vdash \vartheta : \mathcal{T}^B \mid \theta$ and $\Delta;\cdot;\Sigma \vdash \mathsf{Eq}(A,s,t) \mid e$ implies $\Delta;\cdot;\cdot \vdash e\vartheta : B \mid e\theta$.

5.5 Normalisation

Normalisation equates derivations and since **JL**$^\bullet$ internalises its own derivations by means of proof codes, normalisation steps must explicitly relate proof codes in order for SR to hold. Normalisation is modeled as a two step process. First a *principle contraction* is applied, then a series of *permutation conversions* follow. Principle contractions introduce witnesses of derivation compatibility. Permutation conversions standardise derivations by moving these witnesses to the innermost $\Box$ introduction scheme. They permute instances of Eq past any of the inference schemes in $\{\supset\mathsf{I}, \supset\mathsf{E}, \Box\mathsf{E}, \mathsf{Eq}, \mathrm{TI}\}$. Eq just above the left hypothesis of an instance of $\Box\mathsf{I}$ is composed with the trail of the corresponding unit (cf. end of Sec. 5.5.2). In this sense, instances of $\Box\mathsf{I}$ determine the scope of the audited unit.

5.5.1 Principle Contractions

There are three principal contractions (β, $\beta_\Box$ and TI-contraction), the first two of which rely on the substitution principles discussed earlier. The first replaces a derivation of the form:

$$\dfrac{\dfrac{\dfrac{\pi_1}{\Delta;\Gamma_1,a:A;\Sigma_1\vdash B\mid s}}{\Delta;\Gamma_1;\Sigma_1\vdash A\supset B\mid\lambda a^A.s}\supset\mathsf{I}\qquad\dfrac{\pi_2}{\Delta;\Gamma_2;\Sigma_2\vdash A\mid t}}{\Delta;\Gamma_{1,2};\Sigma_{1,2}\vdash B\mid(\lambda a^A.s)\cdot t}\supset\mathsf{E}$$

by the following, where π_3 is a derivation of $\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash B \mid s^a_t$ resulting from π_1 and π_2 and the Substitution Principle for Truth Hypothesis:

$$\dfrac{\pi_3\qquad\dfrac{\dfrac{\pi_1}{\Delta;\Gamma_1,a:A;\Sigma_1\vdash B\mid s}\qquad\dfrac{\pi_2}{\Delta;\Gamma_2;\Sigma_2\vdash A\mid t}}{\Delta;\Gamma_{1,2};\Sigma_{1,2}\vdash\mathsf{Eq}(B,s^a_t,(\lambda a^A.s)\cdot t)\mid\mathfrak{ba}(a^A.s,t)}}{\Delta;\Gamma_{1,2};\Sigma_{1,2}\vdash B\mid(\lambda a^A.s)\cdot t}\mathsf{Eq}$$

The second contraction replaces:

$$\dfrac{\dfrac{\begin{array}{c}\Delta;\cdot;\Sigma\vdash A\mid s\\ \Delta;\cdot;\Sigma\vdash\mathsf{Eq}(A,s,t)\mid e_1\end{array}}{\Delta;\Gamma_1;\Sigma_1\vdash[\![\Sigma.t]\!]A\mid\Sigma.t}\Box\mathsf{I}\qquad\Delta,u:A[\Sigma];\Gamma_2;\Sigma_2\vdash C\mid r}{\Delta;\Gamma_{1,2};\Sigma_{1,2}\vdash C^u_{\Sigma.t}\mid\textsc{let}(u^{A[\Sigma]}.r,\Sigma.t)}\Box\mathsf{E}$$

with the following derivation where π is a derivation of $\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash C^u_{\Sigma.t} \mid t^u_{\Sigma.t}$ resulting from the Substitution Principle for Validity Hypothesis followed by weakening (of Γ_1 and Σ_1) and e_2 is $\mathfrak{bb}(u^{A[\Sigma]}.r,\Sigma.t)$:

$$\dfrac{\pi \qquad \dfrac{\begin{array}{c}\Delta;\cdot;\Sigma \vdash A \mid s \\ \Delta;\cdot;\Sigma \vdash \mathsf{Eq}(A,s,t) \mid e_1 \\ \Delta, u : A[\Sigma];\Gamma_2;\Sigma_2 \vdash C \mid r\end{array}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \mathsf{Eq}(C^u_{\Sigma.t}, r^u_{\Sigma.t}, \text{LET}(u^{A[\Sigma]}.r, \Sigma.t)) \mid e_2}\,\mathsf{Eq}\beta_{\Box}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash C^u_{\Sigma.t} \mid \text{LET}(u^{A[\Sigma]}.r, \Sigma.t)}\,\mathsf{Eq}$$

TI-contraction models audit trail inspection. Consider the following derivation, where $\Sigma_1 \subseteq \Sigma_2$, $\Delta' \subseteq \Delta$ and the branch from the depicted instance of TI in π_1 to its conclusion has no instances of $\Box\mathsf{I}$:

$$\dfrac{\begin{array}{c}\dfrac{\begin{array}{c}\alpha : \mathsf{Eq}(A) \in \Sigma_1 \\ \Delta;\cdot;\cdot \vdash \mathcal{T}^B \mid \theta\end{array}}{\Delta;\Gamma;\Sigma_1 \vdash B \mid \alpha\theta}\,\text{TI} \\ \vdots\ \pi_1 \\ \Delta';\cdot;\Sigma_2 \vdash A \mid s\end{array} \qquad \dfrac{\pi_2}{\Delta';\cdot;\Sigma_2 \vdash \mathsf{Eq}(A,s,t) \mid e}}{\Delta';\Gamma';\Sigma' \vdash [\![\Sigma_2.t]\!]A \mid \Sigma_2.t}\,\Box\mathsf{I}$$

The instance of TI in π_1 is replaced by the following derivation where π_2' is obtained from π_2 by resorting to Lem. 5.4.4 and weakening (Lem. 5.4.1). Also, $\Delta;\cdot;\Sigma_2 \vdash \mathsf{Eq}(A,s,t) \mid e$ is obtained from $\Delta';\cdot;\Sigma_2 \vdash \mathsf{Eq}(A,s,t) \mid e$ by weakening (Lem. 5.4.1).

$$\dfrac{\dfrac{\pi_2'}{\Delta;\Gamma;\Sigma_1 \vdash B \mid e\theta} \qquad \dfrac{\begin{array}{c}\Delta;\cdot;\Sigma_2 \vdash \mathsf{Eq}(A,s,t) \mid e \\ \Delta;\cdot;\cdot \vdash \mathcal{T}^B \mid \theta\end{array}}{\Delta;\Gamma;\Sigma_1 \vdash \mathsf{Eq}(B, e\theta, \alpha\theta) \mid \mathsf{ti}(\theta,\alpha)}\,\text{EqTI}}{\Delta;\Gamma;\Sigma_1 \vdash B \mid \alpha\theta}\,\text{Eq}$$

5.5.2 Permutation Conversions

As for the permutation conversions, they indicate how Eq is permuted past any of the inference schemes in $\{\supset\mathsf{I}, \supset\mathsf{E}, \Box\mathsf{E}, \mathsf{Eq}, \text{TI}\}$. In the first case, Eq permutes past $\supset\mathsf{I}$ by replacing:

$$\dfrac{\dfrac{\dfrac{\pi_1}{\Delta;\Gamma, a : A;\Sigma \vdash B \mid s} \qquad \dfrac{\pi_2}{\Delta;\Gamma, a : A;\Sigma \vdash \mathsf{Eq}(B,s,t) \mid e}}{\Delta;\Gamma, a : A;\Sigma \vdash B \mid t}\,\mathsf{Eq}}{\Delta;\Gamma;\Sigma \vdash A \supset B \mid \lambda a^A.t}\,\supset\mathsf{I}$$

with the following derivation where π_3 is a derivation of $\Delta;\Gamma;\Sigma \vdash A \supset B \mid \lambda a^A.s$ obtained from π_1 and $\supset\mathsf{I}$:

$$\dfrac{\pi_3 \qquad \dfrac{\Delta;\Gamma, a : A;\Sigma \vdash \mathsf{Eq}(B,s,t) \mid e}{\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A \supset B, \lambda a^A.s, \lambda a^A.t) \mid \mathfrak{abC}(a^A.e)}\,\mathsf{EqAbs}}{\Delta;\Gamma;\Sigma \vdash A \supset B \mid \lambda a^A.t}\,\mathsf{Eq}$$

There are two permutation conversions associated to the $\supset\mathsf{E}$ inference scheme depending on whether the instance of Eq is the last scheme in the left or the right hypothesis. In the former the permutation conversion consists in replacing,

$$\dfrac{\dfrac{\begin{array}{c}\Delta;\Gamma_1;\Sigma_1 \vdash A_1 \supset A_2 \mid s \\ \Delta;\Gamma_1;\Sigma_1 \vdash \mathsf{Eq}(A_1 \supset A_2, s, t) \mid e\end{array}}{\Delta;\Gamma_1;\Sigma_1 \vdash A_1 \supset A_2 \mid t}\,\mathsf{Eq} \qquad \Delta;\Gamma_2;\Sigma_2 \vdash A_1 \mid r}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash A_2 \mid t \cdot r}\,\supset\mathsf{E}$$

with the derivation

$$\dfrac{\dfrac{\begin{array}{c}\Delta;\Gamma_1;\Sigma_1 \vdash A_1 \supset A_2 \mid s\\ \Delta;\Gamma_2;\Sigma_2 \vdash A_1 \mid r\end{array}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash A_2 \mid s\cdot r}\supset\mathsf{E} \qquad \dfrac{\pi_1}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \mathsf{Eq}(A_2, s\cdot r, t\cdot r) \mid \mathfrak{apC}(e,\mathfrak{r}(r))}\mathsf{EqApp}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash A_2 \mid t\cdot r}\mathsf{Eq}$$

where π_1 is

$$\dfrac{\Delta;\Gamma_1;\Sigma_1 \vdash \mathsf{Eq}(A_1 \supset A_2, s, t) \mid e \qquad \dfrac{\Delta;\Gamma_2;\Sigma_2 \vdash A_1 \mid r}{\Delta;\Gamma_2;\Sigma_2 \vdash \mathsf{Eq}(A_1, r, r) \mid \mathfrak{r}(r)}\mathsf{EqRefl}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \mathsf{Eq}(A_2, s\cdot r, t\cdot r) \mid \mathfrak{apC}(e,\mathfrak{r}(r))}\mathsf{EqApp}$$

In the latter the permutation conversion consists in replacing,

$$\dfrac{\Delta;\Gamma_1;\Sigma_1 \vdash A_1 \supset A_2 \mid r \qquad \dfrac{\begin{array}{c}\Delta;\Gamma_2;\Sigma_2 \vdash A_1 \mid s\\ \Delta;\Gamma_2;\Sigma_2 \vdash \mathsf{Eq}(A_1, s, t) \mid e\end{array}}{\Delta;\Gamma_2;\Sigma_2 \vdash A_1 \mid t}\mathsf{Eq}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash A_2 \mid r\cdot t}\supset\mathsf{E}$$

with the derivation,

$$\dfrac{\dfrac{\begin{array}{c}\Delta;\Gamma_1;\Sigma_1 \vdash A_1 \supset A_2 \mid r\\ \Delta;\Gamma_2;\Sigma_2 \vdash A_1 \mid s\end{array}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash A_2 \mid r\cdot s}\supset\mathsf{E} \qquad \dfrac{\pi_1}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \mathsf{Eq}(A_2, r\cdot s, r\cdot t) \mid \mathfrak{apC}(\mathfrak{r}(r), e)}\mathsf{EqApp}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash A_2 \mid r\cdot t}\mathsf{Eq}$$

where π_1 is

$$\dfrac{\dfrac{\Delta;\Gamma_1;\Sigma_1 \vdash A_1 \supset A_2 \mid r}{\Delta;\Gamma_1;\Sigma_1 \vdash \mathsf{Eq}(A_1 \supset A_2, r, r) \mid \mathfrak{r}(r)}\mathsf{EqRefl} \qquad \Delta;\Gamma_2;\Sigma_2 \vdash \mathsf{Eq}(A_1, s, t) \mid e}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \mathsf{Eq}(A_2, r\cdot s, r\cdot t) \mid \mathfrak{apC}(\mathfrak{r}(r), e)}\mathsf{EqApp}$$

For the same reasons that there are two permutation conversions associated to $\supset\mathsf{E}$, there are also two associated to $\Box\mathsf{E}$. The first consists in replacing,

$$\dfrac{\dfrac{\begin{array}{c}\Delta;\Gamma_1;\Sigma_1 \vdash [\![\Sigma.s_1]\!]A \mid s_2\\ \Delta;\Gamma_1;\Sigma_1 \vdash \mathsf{Eq}([\![\Sigma.s_1]\!]A, s_2, r) \mid e\end{array}}{\Delta;\Gamma_1;\Sigma_1 \vdash [\![\Sigma.s_1]\!]A \mid r}\mathsf{Eq} \qquad \Delta, u : A[\Sigma];\Gamma_2;\Sigma_2 \vdash C \mid t}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash C^u_{\Sigma.s_1} \mid \textsc{let}(u^{A[\Sigma]}.t, r)}\Box\mathsf{E}$$

with

$$\dfrac{\dfrac{\pi_1}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash C^u_{\Sigma.s_1} \mid q}\Box\mathsf{E} \qquad \dfrac{\pi_2}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \mathsf{Eq}(C^u_{\Sigma.s_1}, q, q') \mid e'}\mathsf{EqLet}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash C^u_{\Sigma.s_1} \mid \textsc{let}(u^{A[\Sigma]}.t, r)}\mathsf{Eq}$$

where q is the proof code $\textsc{let}(u^{A[\Sigma]}.t, s_2)$ and π_1 is:

$$\dfrac{\Delta;\Gamma_1;\Sigma_1 \vdash [\![\Sigma.s_1]\!]A \mid s_2 \quad \Delta, u : A[\Sigma];\Gamma_2;\Sigma_2 \vdash C \mid t}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash C^u_{\Sigma.s_1} \mid \textsc{let}(u^{A[\Sigma]}.t, s_2)}\Box\mathsf{E}$$

and π_2 is the following derivation where q' is $\textsc{let}(u^{A[\Sigma]}.t, r)$, $e' \overset{\text{def}}{=} \mathfrak{leC}(u^{A[\Sigma]}.\mathfrak{r}(t), e)$:

$$\dfrac{\Delta;\Gamma_1;\Sigma_1 \vdash \mathsf{Eq}([\![\Sigma.s_1]\!]A, s_2, r) \mid e \qquad \dfrac{\Delta, u : A[\Sigma];\Gamma_2;\Sigma_2 \vdash C \mid t}{\Delta, u : A[\Sigma];\Gamma_2;\Sigma_2 \vdash \mathsf{Eq}(C, t, t) \mid \mathfrak{r}(t)}\mathsf{EqRefl}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \mathsf{Eq}(C^u_{\Sigma.s_1}, q, q') \mid e'}\mathsf{EqLet}$$

The second permutation conversion associated to $\Box\mathsf{E}$ consists in replacing,

$$\dfrac{\Delta;\Gamma_1;\Sigma_1 \vdash [\![\Sigma.r]\!]A \mid s \qquad \dfrac{\begin{array}{c}\Delta,u:A[\Sigma];\Gamma_2;\Sigma_2\vdash C\mid t_1\\ \Delta,u:A[\Sigma];\Gamma_2;\Sigma_2\vdash \mathsf{Eq}(C,t_1,t_2)\mid e\end{array}}{\Delta,u:A[\Sigma];\Gamma_2;\Sigma_2\vdash C\mid t_2}\,\mathsf{Eq}}{\Delta;\Gamma_{1,2};\Sigma_{1,2}\vdash C^u_{\Sigma.r}\mid \textsc{let}(u^{A[\Sigma]}.t_2,s)}\,\Box\mathsf{E}$$

with the derivation,

$$\dfrac{\dfrac{\Delta;\Gamma_1;\Sigma_1 \vdash [\![\Sigma.r]\!]A \mid s \quad \Delta,u:A[\Sigma];\Gamma_2;\Sigma_2\vdash C\mid t_1}{\Delta;\Gamma_{1,2};\Sigma_{1,2}\vdash C^u_{\Sigma.r}\mid \textsc{let}(u^{A[\Sigma]}.t_1,s)}\,\Box\mathsf{E} \qquad \pi_1}{\Delta;\Gamma_{1,2};\Sigma_{1,2}\vdash C^u_{\Sigma.r}\mid \textsc{let}(u^{A[\Sigma]}.t_2,s)}\,\mathsf{Eq}$$

where, π_1 is

$$\dfrac{\pi_2 \quad \Delta,u:A[\Sigma];\Gamma_2;\Sigma_2\vdash \mathsf{Eq}(C,t_1,t_2)\mid e}{\Delta;\Gamma_{1,2};\Sigma_{1,2}\vdash \mathsf{Eq}(C^u_{\Sigma.r},q,q')\mid \mathfrak{leC}(\mathfrak{r}(s),u^{A[\Sigma]}.e)}\,\mathsf{EqLet}$$

where $q \overset{\text{def}}{=} \textsc{let}(u^{A[\Sigma]}.t_1,s)$ and $q' \overset{\text{def}}{=} \textsc{let}(u^{A[\Sigma]}.t_2,s)$ and π_2 is:

$$\dfrac{\Delta;\Gamma_1;\Sigma_1\vdash [\![\Sigma.r]\!]A\mid s}{\Delta;\Gamma_1;\Sigma_1\vdash \mathsf{Eq}([\![\Sigma.r]\!]A,s,s)\mid \mathfrak{r}(s)}\,\mathsf{EqRefl}$$

Two permutation conversions remain, namely permutation with Eq itself and the case where proof code passes from the left hypothesis in an introduction of the modality to the right hypothesis. These two cases are given in Fig. 5.4. In the last one c is some trail constructor and

- $\theta'(c) \overset{\text{def}}{=} r$ and $\theta'(d) \overset{\text{def}}{=} \theta(d)$, for all $d \in \mathcal{C}$ s.t. $d \neq c$; and
- $e_c \overset{\text{def}}{=} e$ and $e_d \overset{\text{def}}{=} \mathfrak{r}(\theta(d))$, for all $d \neq c$.

$$\dfrac{\dfrac{\begin{array}{c}\Delta;\Gamma;\Sigma\vdash A\mid r\\ \Delta;\Gamma;\Sigma\vdash \mathsf{Eq}(A,r,s)\mid e_1\end{array}}{\Delta;\Gamma;\Sigma\vdash A\mid s}\,\mathsf{Eq} \qquad \Delta;\Gamma;\Sigma\vdash \mathsf{Eq}(A,s,t)\mid e_2}{\Delta;\Gamma;\Sigma\vdash A\mid t}\,\mathsf{Eq} \qquad \dfrac{\Delta;\Gamma;\Sigma\vdash A\mid r \qquad \dfrac{\begin{array}{c}\Delta;\Gamma;\Sigma\vdash \mathsf{Eq}(A,r,s)\mid e_1\\ \Delta;\Gamma;\Sigma\vdash \mathsf{Eq}(A,s,t)\mid e_2\end{array}}{\Delta;\Gamma;\Sigma\vdash \mathsf{Eq}(A,r,t)\mid \mathfrak{t}(e_1,e_2)}\,\mathsf{EqTrans}}{\Delta;\Gamma;\Sigma\vdash A\mid t}\,\mathsf{Eq}$$

$$\dfrac{\begin{array}{c}\Delta;\cdot;\cdot\vdash \mathcal{T}^B\mid\theta\\ \alpha:\mathsf{Eq}(A)\in\Sigma\end{array} \qquad \dfrac{\begin{array}{c}\Delta;\cdot;\cdot\vdash \mathcal{T}^B(c)\mid r\\ \Delta;\cdot;\cdot\vdash \mathsf{Eq}(\mathcal{T}^B(c),r,\theta(c))\mid e\end{array}}{\Delta;\cdot;\cdot\vdash \mathcal{T}^B(c)\mid\theta(c)}\,\mathsf{Eq}}{\Delta;\Gamma;\Sigma\vdash B\mid\alpha\theta}\,\text{TI} \qquad \dfrac{\dfrac{\Delta;\cdot;\cdot\vdash \mathcal{T}^B\mid\theta'}{\Delta;\Gamma;\Sigma\vdash B\mid\alpha\theta'}\,\text{TI} \qquad \dfrac{\Delta;\cdot;\cdot\vdash \mathsf{Eq}(\mathcal{T}^B,\theta',\theta)\mid e_i}{\Delta;\Gamma;\Sigma\vdash \mathsf{Eq}(B,\alpha\theta',\alpha\theta)\mid \mathfrak{rpC}(\overline{e})}\,\textsc{EqRpl}}{\Delta;\Gamma;\Sigma\vdash B\mid\alpha\theta}\,\mathsf{Eq}$$

Fig. 5.4. Permutation conversions for Eq and TI

Finally, we have the conversion that fuses Eq just above the left hypothesis in an instance of $\Box\mathsf{I}$ with the trail of the corresponding unit is also coined, by abuse of language, permutation conversion. It replaces:

$$\dfrac{\dfrac{\Delta;\cdot;\Sigma\vdash A\mid r \quad \Delta;\cdot;\Sigma\vdash \mathsf{Eq}(A,r,s)\mid e_1}{\Delta;\cdot;\Sigma\vdash A\mid s}\,\mathsf{Eq} \qquad \Delta;\cdot;\Sigma\vdash \mathsf{Eq}(A,s,t)\mid e_2}{\Delta;\Gamma;\Sigma'\vdash [\![\Sigma.t]\!]A\mid \Sigma.t}\,\Box\mathsf{I}$$

with:

$$\dfrac{\Delta; \cdot; \Sigma \vdash A \mid r \qquad \dfrac{\Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, r, s) \mid e_1 \quad \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e_2}{\Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, r, t) \mid \mathsf{t}(e_1, e_2)}\,\mathsf{EqTrans}}{\Delta; \Gamma; \Sigma' \vdash [\![\Sigma.t]\!]A \mid \Sigma.t}\,\Box\mathsf{I}$$

5.6 Term Assignment

Computation by normalisation is non-confluent, as one might expect (audit trail inspection affects computation), hence a strategy is required. This section introduces the call-by-value $\lambda^{\mathfrak{h}}$-calculus. It is obtained via a term assignment for **JL**$^\bullet$. Recall from Sec. 5.4 that ϑ in TTI is a mapping from trail constructors to terms. The syntax of $\lambda^{\mathfrak{h}}$ terms is:

$$\begin{array}{rll}
M ::= & a & \text{term variable} \\
\mid & \lambda a^A.M & \text{abstraction} \\
\mid & M\,M & \text{application} \\
\mid & \langle u; \sigma \rangle & \text{audited unit variables} \\
\mid & !^{\Sigma}_{e} M & \text{audited unit} \\
\mid & \text{LET}\, u^{A[\Sigma]} \leftarrow M \,\text{IN}\, M & \text{audited unit composition} \\
\mid & \alpha\vartheta & \text{trail inspection} \\
\mid & e \rhd M & \text{derived term}
\end{array}$$

We occasionally drop the type decoration in audited unit composition for readability. Since terms may be decorated with trails, substitution (both for truth and validity hypothesis) replaces free occurrences of variables with both terms and evidence. We write $M^a_{N,t}$ for substitution of truth variables and $M^u_{\Sigma.(N,t,e)}$ for substitution of validity variables (similar notions apply to substitution in propositions, proof code and trails). Note that "$\Sigma.$" in $\Sigma.(N, t, e)$ binds all free occurrences of trail variables from Σ which occur in N, t and e.

$$\begin{array}{rcl}
a^a_{N,s} & \stackrel{\text{def}}{=} & N \\
b^a_{N,s} & \stackrel{\text{def}}{=} & b \\
(\lambda b^A.M)^a_{N,s} & \stackrel{\text{def}}{=} & \lambda b^A.M^a_{N,s} \\
(P\,Q)^a_{N,s} & \stackrel{\text{def}}{=} & P^a_{N,s}\,Q^a_{N,s} \\
\langle u; \sigma \rangle^a_{N,s} & \stackrel{\text{def}}{=} & \langle u; \sigma \rangle \\
(!^{\Sigma}_{e} M)^a_{N,s} & \stackrel{\text{def}}{=} & !^{\Sigma}_{e} M \\
(\text{LET}\, u \leftarrow M_1 \,\text{IN}\, M_2)^a_{N,s} & \stackrel{\text{def}}{=} & \text{LET}\, u \leftarrow {M_1}^a_{N,s} \,\text{IN}\, {M_2}^a_{N,s} \\
(\alpha\theta)^a_{N,s} & \stackrel{\text{def}}{=} & \alpha\theta \\
(e \rhd M)^a_{N,s} & \stackrel{\text{def}}{=} & e^a_s \rhd M^a_{N,s}
\end{array}
\qquad
\begin{array}{rcl}
b^u_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & b \\
(\lambda b^A.M)^u_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & \lambda b^A.M^u_{\Sigma.(N,t,e)} \\
(P\,Q)^u_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & P^u_{\Sigma.(N,t,e)}\,Q^u_{\Sigma.(N,t,e)} \\
\langle u; \sigma \rangle^u_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & e\sigma \rhd N\sigma \\
\langle v; \sigma \rangle^u_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & \langle v; \sigma \rangle \\
(!^{\Sigma'}_{e'} M)^u_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & !^{\Sigma'}_{{e'}^u_{\Sigma.t}} M^u_{\Sigma.(N,t,e)} \\
(\text{LET}\, v \leftarrow P \,\text{IN} & \stackrel{\text{def}}{=} & \text{LET}\, v \leftarrow P^u_{\Sigma.(N,t,e)} \,\text{IN} \\
Q)^u_{\Sigma.(N,t,e)} & & Q^u_{\Sigma.(N,t,e)} \\
(\alpha\vartheta)^u_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & \alpha\vartheta^u_{\Sigma.(N,t,e)} \\
(e' \rhd M)^u_{\Sigma.(N,t,e)} & \stackrel{\text{def}}{=} & {e'}^u_{\Sigma.t} \rhd M^u_{\Sigma.(N,t,e)}
\end{array}$$

These definitions rely on $s^u_{\Sigma.t}$ which traverses the structure of s replacing $\langle u; \sigma \rangle^u_{\Sigma.s}$ with $s\sigma$ and $e^u_{\Sigma.t}$ which traverses the structure of e until it reaches one of $\mathfrak{r}(r_1), \mathfrak{ba}(a^A.r_1, r_2)$ or $\mathfrak{bb}(v^{A[\Sigma']}.r_1, \Sigma'.r_2)$ in which case it resorts to substitution over the r_is. Of particular interest is the fourth defining clause of the definition of $M^u_{\Sigma.(N,t,e)}$. Note how it substitutes $\langle u; \sigma \rangle$ with $e\sigma \rhd N\sigma$, thus (1) propagating the history of N and (2) rewiring the trail variables of N so that they make sense in the new host unit.

5.6.1 Typing Schemes and Operational Semantics

The typing judgement $\Delta; \Gamma; \Sigma \vdash M : A \mid s$ is defined by means of the *typing schemes*. These are obtained by decorating the inference schemes of Fig. 5.2 with terms. A term M is said to be *typable* if there exists $\Delta, \Gamma, \Sigma, A, s$ s.t. $\Delta; \Gamma; \Sigma \vdash M : A \mid s$ is derivable. The typing schemes are presented in Fig. 5.5. Note that the scheme TEQ incorporates the trail e into the term assignment.

$$\frac{a : A \in \Gamma}{\Delta; \Gamma; \Sigma \vdash a : A \mid a}\,\textsc{TVar} \qquad \frac{\Delta; \Gamma, a : A; \Sigma \vdash M : B \mid s}{\Delta; \Gamma; \Sigma \vdash \lambda a^{A}.M : A \supset B \mid \lambda a^{A}.s}\,\textsc{TAbs}$$

$$\frac{\begin{array}{c}\Delta; \Gamma_1; \Sigma_1 \vdash M : A \supset B \mid s \\ \Delta; \Gamma_2; \Sigma_2 \vdash N : A \mid t\end{array}}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash M\,N : B \mid s \cdot t}\,\textsc{TApp} \qquad \frac{u : A[\Sigma] \in \Delta \quad \Sigma\sigma \subseteq \Sigma'}{\Delta; \cdot; \Sigma' \vdash \langle u; \sigma \rangle : A \mid \langle u; \sigma \rangle}\,\textsc{TmVar}$$

$$\frac{\begin{array}{c}\Delta; \cdot; \Sigma \vdash M : A \mid s \\ \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e\end{array}}{\Delta; \Gamma; \Sigma' \vdash !^{\Sigma}_{e} M : \llbracket \Sigma.t \rrbracket A \mid \Sigma.t}\,\textsc{TBox} \qquad \frac{\begin{array}{c}\Delta; \Gamma_1; \Sigma_1 \vdash M : \llbracket \Sigma.r \rrbracket A \mid s \\ \Delta, u : A[\Sigma]; \Gamma_2; \Sigma_2 \vdash N : C \mid t\end{array}}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash \begin{array}{c}\textsc{let}\, u^{A[\Sigma]} \leftarrow M \,\textsc{in}\, N : C^{u}_{\Sigma.r} \\ \textsc{let}(u^{A[\Sigma]}.t, s)\end{array} \mid}\,\textsc{TLet}$$

$$\frac{\begin{array}{c}\alpha : \mathsf{Eq}(A) \in \Sigma \\ \Delta; \cdot; \cdot \vdash \vartheta : \mathcal{T}^{B} \mid \theta\end{array}}{\Delta; \Gamma; \Sigma \vdash \alpha\vartheta : B \mid \alpha\theta}\,\textsc{TTI} \qquad \frac{\begin{array}{c}\Delta; \Gamma; \Sigma \vdash M : A \mid s \\ \Delta; \Gamma; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e\end{array}}{\Delta; \Gamma; \Sigma \vdash e \rhd M : A \mid t}\,\textsc{TEq}$$

Fig. 5.5. Typing schemes for $\lambda^{\mathfrak{h}}$

The operational semantics of $\lambda^{\mathfrak{h}}$ is specified by a binary relation of *typed* terms called *reduction*. We say that a term M reduces to a term N if the *reduction* judgement $\Delta; \Gamma; \Sigma \vdash M \to N : A \mid s$ is derivable. In most cases we simply write $M \to N$ for the sake of readability. In order to define reduction we first introduce two intermediate notions, namely *principle reduction* ($M \mapsto N$) and *permutation reduction* ($M \leadsto N$). The former corresponds to principle contraction and the latter to permutation conversions of the normalisation procedure. The set of *values* are standard except for $!^{\Sigma}_{e} V$ (audited units with fully evaluated bodies are also values):

$$\begin{array}{lll} \textit{Values} & V & ::= a \mid \langle u; \sigma \rangle \mid \lambda a^{A}.M \mid !^{\Sigma}_{e} V \\ \textit{Value replacement} & \vartheta_V & ::= \{c_1/V_1, \ldots, c_{10}/V_{10}\} \end{array}$$

The expressions c_i, $i \in 1..10$, abbreviate each of the trail constructors introduced in Sec. 5.4. In the sequel of this section we write ϑ rather than ϑ_V. *Evaluation contexts* are represented with letters $\mathcal{E}, \mathcal{E}'$, etc:

$$\begin{array}{rl} \textit{Eval. contexts}\ \mathcal{E} ::= & \Box \mid \mathcal{E}\ M \mid V\ \mathcal{E} \mid \textsc{let}(u^{A[\Sigma]}.M, \mathcal{E}) \mid !^{\Sigma}_{e}\mathcal{E} \\ \mid & \alpha\{c_1/V_1, \ldots, c_j/V_j, c_{j+1}/\mathcal{E}, \ldots\} \\ \mathcal{F} ::= & \Box \mid \mathcal{F}\ M \mid V\ \mathcal{F} \mid \textsc{let}(u^{A[\Sigma]}.M, \mathcal{F}) \end{array}$$

Definition 5.6.1 (Principle Reduction) Principle reduction, $\mapsto$, is defined by the following reduction scheme:

$$\frac{M \rightharpoonup N}{\mathcal{E}[M] \mapsto \mathcal{E}[N]}$$

where $\rightharpoonup$ is the union of the following binary relations:

$$\frac{\Delta; \Gamma_1, a : A; \Sigma_1 \vdash M : B \mid s \quad \Delta; \Gamma_2; \Sigma_2 \vdash V : A \mid t}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash (\lambda a^{A}.M)\ V \rightharpoonup \mathfrak{ba}(a^{A}.s, t) \rhd M^{a}_{V,t} : B \mid (\lambda a^{A}.s) \cdot t}\,\beta^{V}$$

$$\frac{\Delta; \cdot; \Sigma \vdash V : A \mid r \quad \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, r, s) \mid e_1 \quad \Delta, u : A[\Sigma]; \Gamma_2; \Sigma_2 \vdash N : C \mid t}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash O \rightharpoonup P : C^{u}_{\Sigma.s} \mid \textsc{let}(u^{A[\Sigma]}.t, \Sigma.s)}\,\beta^{V}_{\Box}$$

$$\frac{\alpha : \mathsf{Eq}(A) \in \Sigma \quad \Delta'; \cdot; \cdot \vdash \vartheta : \mathcal{T}^{B} \mid \theta \quad \Delta \subseteq \Delta' \quad \Delta; \cdot; \Sigma \vdash \mathcal{F}[\alpha\vartheta] : A \mid r \quad \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, r, s) \mid e}{\Delta; \Gamma; \Sigma' \vdash !^{\Sigma}_{e}\mathcal{F}[\alpha\vartheta] \rightharpoonup !^{\Sigma}_{e}\mathcal{F}[\mathsf{ti}(\theta, \alpha) \rhd e\vartheta] : \llbracket \Sigma.s \rrbracket A \mid \Sigma.s}\,\mathcal{I}^{V}$$

$$
\begin{array}{rcl}
(e \rhd M)\ N & \rightsquigarrow & \mathfrak{apC}(e, \mathfrak{r}(t)) \rhd (M\ N) \\
M\ (e \rhd N) & \rightsquigarrow & \mathfrak{apC}(\mathfrak{r}(t), e) \rhd (M\ N) \\
\lambda a^A.(e \rhd M) & \rightsquigarrow & \mathfrak{abC}(a.e) \rhd (\lambda a^A.M) \\
\text{LET}\, u \leftarrow (e \rhd M)\, \text{IN}\, N & \rightsquigarrow & \mathfrak{leC}(u.e, \mathfrak{r}(t)) \rhd (\text{LET}\, u \leftarrow M\, \text{IN}\, N) \\
\text{LET}\, u \leftarrow M\, \text{IN}\, (e \rhd N) & \rightsquigarrow & \mathfrak{leC}(u.\mathfrak{r}(s), e) \rhd (\text{LET}\, u \leftarrow M\, \text{IN}\, N) \\
!^{\Sigma}_{e_2}(e_1 \rhd M) & \rightsquigarrow & !^{\Sigma}_{\mathfrak{t}(e_1, e_2)} M \\
e_1 \rhd (e_2 \rhd M) & \rightsquigarrow & \mathfrak{t}(e_1, e_2) \rhd M
\end{array}
$$

$$
\begin{array}{rcl}
\mathfrak{t}(\mathfrak{apC}(e_1, e_2), \mathfrak{apC}(e_3, e_4)) & \rightsquigarrow & \mathfrak{apC}(\mathfrak{t}(e_1, e_3), \mathfrak{t}(e_2, e_4)) \\
\mathfrak{t}(\mathfrak{abC}(a.e_1), \mathfrak{abC}(a.e_2)) & \rightsquigarrow & \mathfrak{abC}(a.\mathfrak{t}(e_1, e_2)) \\
\mathfrak{t}(\mathfrak{leC}(u.e_1, e_2), \mathfrak{leC}(u.e_3, e_4)) & \rightsquigarrow & \mathfrak{leC}(u.\mathfrak{t}(e_1, e_3), \mathfrak{t}(e_2, e_4)) \\
\mathfrak{t}(\mathfrak{r}(s), e) & \rightsquigarrow & e \\
\mathfrak{t}(e, \mathfrak{r}(t)) & \rightsquigarrow & e \\
\mathfrak{t}(\mathfrak{t}(e_1, e_2), e_3) & \rightsquigarrow & \mathfrak{t}(e_1, \mathfrak{t}(e_2, e_3)) \\
\mathfrak{t}(\mathfrak{apC}(e_1, e_2), \mathfrak{t}(\mathfrak{apC}(e_3, e_4), e_5)) & \rightsquigarrow & \mathfrak{t}(\mathfrak{apC}(\mathfrak{t}(e_1, e_3), \mathfrak{t}(e_2, e_4)), e_5) \\
\mathfrak{t}(\mathfrak{abC}(a.e_1), \mathfrak{t}(\mathfrak{abC}(a.e_2), e_3)) & \rightsquigarrow & \mathfrak{t}(\mathfrak{abC}(a.\mathfrak{t}(e_1, e_2)), e_3) \\
\mathfrak{t}(\mathfrak{leC}(u.e_1, e_2), \mathfrak{t}(\mathfrak{leC}(u.e_3, e_4), e_5)) & \rightsquigarrow & \mathfrak{t}(\mathfrak{leC}(u.\mathfrak{t}(e_1, e_3), \mathfrak{t}(e_2, e_4)), e_5)
\end{array}
$$

Fig. 5.6. Permutation reduction schemes

where, in $\beta^V_\Box$, $O \stackrel{\text{def}}{=} \text{LET}\, u^{A[\Sigma]} \leftarrow !^{\Sigma}_{e} V\, \text{IN}\, N$ and $P \stackrel{\text{def}}{=} \mathfrak{bb}(u^{A[\Sigma]}.t, \Sigma.s) \rhd N^u_{\Sigma.(V,s,e)}$.

Note that reduction under the audited unit constructor is allowed. Contexts $\mathcal{F}$ are required for defining $\mathcal{I}$(nspection), the principle reduction axiom for trail inspection. It differs from $\mathcal{E}$ by not allowing holes under the audited unit constructor. Each principle reduction scheme produces a trail of its execution. In accordance with our discussion in the introduction, $\beta^V_\Box$ replaces all occurrences of $\langle u; \sigma \rangle$ with $e\sigma \rhd V\sigma$, correctly: (1) preserving trails and (2) rewiring trail variables so that they now refer to their host audited computation unit.

Regarding permutation reduction, the original schemes obtained from the normalisation procedure are the contextual closure of the first group of rules depicted in Fig. 5.6, where type decorations in compatibility codes have been omitted for the sake of readability. These schemes are easily proven to be terminating. However, they are not confluent. As an example, consider the term $(e_1 \rhd M)\,(e_2 \rhd N)$. Application of the first rule yields $\mathfrak{apC}(e_1, \mathfrak{r}(t_2)) \rhd (M\ (e_2 \rhd N))$ where t_2 is the proof code of the type derivation of $e_2 \rhd N$ in $(e_1 \rhd M)\,(e_2 \rhd N)$. Likewise, application of the second yields $\mathfrak{apC}(\mathfrak{r}(t_1), e_2) \rhd ((e_1 \rhd M)N)$ where t_1 is the proof code of the type derivation of $e_1 \rhd M$ in $(e_1 \rhd M)\,(e_2 \rhd N)$. The reader may verify that these terms are not joinable. As a consequence we complete these schemes with those in the second group depicted in Fig. 5.6.

Definition 5.6.2 (Permutation reduction) Permutation reduction, $\rightsquigarrow$, is defined by the contextual closure of the reduction axioms of Fig. 5.6

Proposition 5.6.1 $\rightsquigarrow$ is terminating and confluent.

Termination may be proved automatically by using AProVE [59]. Confluence follows by checking local confluence and resorting to Newman's Lemma.

Remark 3 The fact that these reduction schemes are defined over typed terms is crucial for confluence. Indeed, the system is not confluent over the set of untyped terms. For example, $\mathfrak{t}(\mathfrak{r}(s), \mathfrak{r}(t))$ reduces to both $\mathfrak{r}(s)$ and $\mathfrak{r}(t)$. However, in a typed setting $\mathfrak{t}(\mathfrak{r}(s), \mathfrak{r}(t))$ typable implies $s = t$.

Definition 5.6.3 (Reduction) Let $\rightarrowtail$ stand for permutation reduction to (the unique) normal form. Reduction ($\to$) is defined over terms in permutation-reduction normal form as $\mapsto \circ \rightarrowtail$, where $\circ$ denotes relation composition.

We summarise the relations introduced in this section:

$\mapsto$	principle reduction
$\leadsto$	permutation reduction
$\rightarrowtail$	permutation reduction to normal form
$\to$	reduction

5.6.2 Safety

We now address safety of reduction w.r.t. the type system. This involves proving SR and Progress. SR follows from the fact that the reduction schemes originate from proof normalisation. The exception are the second group of schemes of Fig. 5.6 for which type preservation may also be proved separately.

Proposition 5.6.2 (Subject Reduction) $\Delta; \Gamma; \Sigma \vdash M : A \mid s$ and $M \to N$ implies $\Delta; \Gamma; \Sigma \vdash N : A \mid s$.

Before addressing Progress we introduce some auxiliary notions. A term is *inspection-blocked* if it is of the form $\mathcal{F}[\alpha\theta]$. A term M is *tv-closed* if $\mathsf{fvT}(M) = \mathsf{fvV}(M) = \emptyset$. It is *closed* if it is tv-closed and $\mathsf{fvTrl}(M) = \emptyset$.

Lemma 5.6.3 (Canonical forms) Assume $\cdot; \cdot; \Sigma \vdash V : A \mid s$. Then,

1. If $A = A_1 \supset A_2$, then $V = \lambda a^{A_1}.M$ for some a, M.
2. If $A = [\![\Sigma'.t]\!]A_1$, then $V = !_e^{\Sigma'} V'$ for some e, V'.

Proposition 5.6.4 Suppose M is in permutation reduction-normal form, is typable and tv-closed. Then (1) M is a value or; (2) there exists N s.t. $M \mapsto N$ or; (3) M is inspection-blocked.

Since a closed term cannot be inspection-blocked:

Corollary 1 Progress Suppose M is in permutation reduction normal form, is typable and closed. Then either M is a value or there exists N s.t. $M \to N$.

5.7 Strong Normalisation

We address SN for reduction. In fact, we shall prove SN for a restriction of reduction. The restriction consists in requiring that M in the principle reduction axiom $\beta_{\Box}^{V}$ not have occurrences of the audited computation unit constructor "!"[6]. In order to develop our proof, we introduce the notion of *full reduction* which lifts the value restriction in β^V and $\beta_{\Box}^{V}$; and allows more general evaluation contexts for all three axioms.

Definition 5.7.1 (Full Principle Reduction) Full principle reduction, $\overset{f}{\mapsto}$, is defined by the following reduction scheme:

$$\frac{M \overset{f}{\rightharpoonup} N}{\mathcal{D}[M] \overset{f}{\mapsto} \mathcal{D}[N]}$$

[6] We currently have no proof for unrestricted reduction, however we believe the result should hold.

$$
\begin{array}{llll}
\mathcal{D} ::= \Box \,|\, \lambda a^A.\mathcal{D} \,|\, \mathcal{D}\ M \,|\, M\ \mathcal{D} & & \mathcal{C} ::= \Box \,|\, \lambda a^A.\mathcal{C} \,|\, \mathcal{C}\ M \,|\, M\ \mathcal{C} \\
\quad |\ \ \textsc{let}\, u^{A[\Sigma]} \leftarrow \mathcal{D}\, \textsc{in}\, M & & \quad |\ \ \textsc{let}\, u^{A[\Sigma]} \leftarrow \mathcal{C}\, \textsc{in}\, M \\
\quad |\ \ \textsc{let}\, u^{A[\Sigma]} \leftarrow M\, \textsc{in}\, \mathcal{D} \,|\, !^{\Sigma}_{e}\mathcal{D} \,|\, e \rhd \mathcal{D} & & \quad |\ \ \textsc{let}\, u^{A[\Sigma]} \leftarrow M\, \textsc{in}\, \mathcal{C} \\
\quad |\ \ \alpha\{c_1/M_1, \ldots, c_j/M_j, c_{j+1}/\mathcal{D}, \ldots\} & & \quad |\ \ e \rhd \mathcal{C}
\end{array}
$$

Fig. 5.7. Full principle reduction

where $\xrightarrow{f}$ is the union of the following binary relations:

$$
\frac{\Delta; \Gamma_1, a : A; \Sigma_1 \vdash M : B \mid s \quad \Delta; \Gamma_2; \Sigma_2 \vdash N : A \mid t}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash (\lambda a^A.M)\ N \xrightarrow{f} \mathfrak{ba}(a^A.s, t) \rhd M^a_{N,t} : B \mid (\lambda a^A.s) \cdot t}\ \beta
$$

$$
\frac{\Delta; \cdot; \Sigma \vdash M : A \mid r \quad M \text{ is ! free} \quad \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, r, s) \mid e_1 \quad \Delta, u : A[\Sigma]; \Gamma_2; \Sigma_2 \vdash N : C \mid t}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash O \xrightarrow{f} P : C^u_{\Sigma.s} \mid \textsc{let}(u^{A[\Sigma]}.t, \Sigma.s)}\ \beta_\Box
$$

$$
\frac{\alpha : \mathsf{Eq}(A) \in \Sigma \quad \Delta'; \cdot; \cdot \vdash \vartheta : \mathcal{T}^B \mid \theta \quad \Delta \subseteq \Delta' \quad \Delta; \cdot; \Sigma \vdash \mathcal{C}[\alpha\vartheta] : A \mid r \quad \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, r, s) \mid e}{\Delta; \Gamma; \Sigma' \vdash !^{\Sigma}_{e}\mathcal{C}[\alpha\vartheta] \xrightarrow{f} !^{\Sigma}_{e}\mathcal{C}[\mathsf{ti}(\theta, \alpha) \rhd e\vartheta] : [\![\Sigma.s]\!]A \mid \Sigma.s}\ \mathcal{I}
$$

where, in $\beta_\Box$, $O \stackrel{\text{def}}{=} \textsc{let}\, u^{A[\Sigma]} \leftarrow !^{\Sigma}_{e} M\, \textsc{in}\, N$ and $P \stackrel{\text{def}}{=} \mathfrak{bb}(u^{A[\Sigma]}.t, \Sigma.s) \rhd N^u_{\Sigma.(M,s,e)}$.

Definition 5.7.2 (Full reduction) Full reduction, $\xrightarrow{f}$, is defined as the union of *full principle reduction* ($\overset{f}{\mapsto}$) and permutation reduction ($\rightsquigarrow$).

In the sequel, we write $\overset{rf}{\mapsto}$ for the abovementioned restricted notion of reduction. The proof is by contradiction and is developed in two steps. The first shows that an infinite $\overset{f}{\mapsto} \cup \rightsquigarrow$ reduction sequence must include an infinite number of $\overset{f}{\mapsto}_{\mathcal{I}}$ steps. The second, that $\overset{f}{\mapsto}_{\mathcal{I}}$ is SN.

5.7.1 Step 1

We first note that $\overset{f}{\mapsto}_{\beta,\beta_\Box}$ is SN. This can be proved by defining a translation $\mathcal{S}(\bullet)$ on $\lambda^{\natural}$ types that "forget" the modal connective and a similar translation from terms in $\lambda^{\natural}$ to terms of the simply typed lambda calculus (with constants) such that: (1) it preserves typability; and (2) it maps full reduction to reduction in the simply typed lambda calculus. Since we already know that $\rightsquigarrow$ is SN and that reduction in the simply typed lambda calculus is SN, our result shall follow. For a proposition A, $\mathcal{S}(A)$ produces a type of the simply typed lambda calculus by "forgetting" the modal type constructor; for contexts it behaves homomorphically producing multisets of labeled hypothesis:

$$
\begin{array}{rlrl}
\mathcal{S}(P) & \stackrel{\text{def}}{=} P & \mathcal{S}(\cdot) & \stackrel{\text{def}}{=} \cdot \\
\mathcal{S}(A \supset B) & \stackrel{\text{def}}{=} \mathcal{S}(A) \supset \mathcal{S}(B) & \mathcal{S}(\Gamma, a : A) & \stackrel{\text{def}}{=} \mathcal{S}(\Gamma), a : \mathcal{S}(A) \\
\mathcal{S}([\![\Sigma.s]\!]A) & \stackrel{\text{def}}{=} \mathcal{S}(A) & \mathcal{S}(\Delta, u : A[\Sigma]) & \stackrel{\text{def}}{=} \mathcal{S}(\Delta), u : \mathcal{S}(A)
\end{array}
$$

The *decoration* of a typed term is the expression obtained from replacing every occurrence of the trail lookup expression $\alpha\vartheta$ by $\alpha^B\vartheta$, where B is the result type of the lookup. We assume we have a constant c_B for each type B. For a term M, $\mathcal{S}(M)$ produces a term of the simply typed lambda calculus:

$$\begin{array}{rcl}
\mathcal{S}(a) & \stackrel{\text{def}}{=} & a \\
\mathcal{S}(\lambda a^A.M) & \stackrel{\text{def}}{=} & \lambda a^{\mathcal{S}(A)}.\mathcal{S}(M) \\
\mathcal{S}(M\,N) & \stackrel{\text{def}}{=} & \mathcal{S}(M)\,\mathcal{S}(N) \\
\mathcal{S}(\langle u;\sigma\rangle) & \stackrel{\text{def}}{=} & u \\
\mathcal{S}(!^{\Sigma}_{e}M) & \stackrel{\text{def}}{=} & \mathcal{S}(M) \\
\mathcal{S}(\text{LET}\, u^{A[\Sigma]} \leftarrow M \text{ IN } N) & \stackrel{\text{def}}{=} & (\lambda u^{\mathcal{S}(A)}.\mathcal{S}(N))\,\mathcal{S}(M) \\
\mathcal{S}(\alpha^B\vartheta) & \stackrel{\text{def}}{=} & (\lambda a : \mathcal{S}(\mathcal{T}^B).c_B)\,\mathcal{S}(\vartheta) \\
\mathcal{S}(e \rhd M) & \stackrel{\text{def}}{=} & \mathcal{S}(M)
\end{array}$$

where $\mathcal{S}(\mathcal{T}^B)$ abbreviates the product type $\langle \mathcal{S}(\mathcal{T}^B(c_1)), \ldots, \mathcal{S}(\mathcal{T}^B(c_{10}))\rangle$ and $\mathcal{S}(\vartheta)$ abbreviates the product term $\langle \mathcal{S}(\vartheta(c_1)), \ldots, \mathcal{S}(\vartheta(c_{10}))\rangle$.

Lemma 5.7.1 If $\Delta;\Gamma;\Sigma \vdash M : A \mid s$, then $\mathcal{S}(\Delta), \mathcal{S}(\Gamma) \vdash \mathcal{S}(M) : \mathcal{S}(A)$.

The following result is verified by noticing that $\mathcal{S}(\bullet)$ erases both evidence and the modal term constructor in terms.

Lemma 5.7.2 If $M \rightsquigarrow N$, then $\mathcal{S}(M) = \mathcal{S}(N)$.

The translation function $\mathcal{S}(\bullet)$ permutes with both truth and validity substitution. Note that $\mathcal{S}(M)^a_{\mathcal{S}(N)}$ and $\mathcal{S}(M)^u_{\mathcal{S}(N)}$ below, is substitution in the simply typed lambda calculus. The proofs of these items is by induction on the structure of M; the third item resorts to the first one.

Lemma 5.7.3 1. $\mathcal{S}(M\sigma) = \mathcal{S}(M)$.
2. $\mathcal{S}(M^a_{N,t}) = \mathcal{S}(M)^a_{\mathcal{S}(N)}$.
3. $\mathcal{S}(M^u_{\Sigma.(N,t,e)}) = \mathcal{S}(M)^u_{\mathcal{S}(N)}$.

Lemma 5.7.4 If $M \overset{f}{\mapsto}_{\beta,\beta_\Box} N$, then $\mathcal{S}(M) \overset{\lambda^{\supset,\times}}{\to} \mathcal{S}(N)$.

The following result is a consequence of Lem. 5.7.2 and Lem. 5.7.4.

Corollary 2 If $M \overset{\beta,\beta_\Box}{\to} N$, then $\mathcal{S}(M) \overset{\lambda^{\supset,\times}}{\to} \mathcal{S}(N)$.

We may thus conclude with the following result.

Proposition 5.7.5 $\overset{f}{\mapsto}_{\beta,\beta_\Box} \cup \rightsquigarrow$ is SN.

Therefore, an infinite $\overset{f}{\mapsto} \cup \rightsquigarrow$ reduction sequence must include an infinite number of $\overset{f}{\mapsto}_{\mathcal{I}}$ steps.

5.7.2 Step 2

Next we show that for $\overset{rf}{\mapsto}$ this is not possible. More precisely, we show that in an infinite $\overset{rf}{\mapsto} \cup \rightsquigarrow$ reduction sequence, there can only be a finite number of $\overset{f}{\mapsto}_{\mathcal{I}}$ steps. This entails:

Proposition 5.7.6 $\overset{rf}{\mapsto} \cup \rightsquigarrow$ is SN. Hence $\lambda^{\natural}$, with the same restriction, is SN.

We now address the proof of the main lemma on which Prop. 5.7.6 relies (Lem. 5.7.10). We introduce weight functions which strictly decrease by each application of a $\overset{f}{\mapsto}_{\mathcal{I}}$-step and which decreases with each application of a $\overset{rf}{\mapsto}_{\beta,\beta_\Box}$-step or $\rightsquigarrow$-step. A word on notation: $\langle\!\langle\ \rangle\!\rangle$ is the empty multiset; $\uplus$ is multiset union; and $n \uplus \mathcal{M}$ is the union of the multiset $\langle\!\langle n \rangle\!\rangle$ and $\mathcal{M}$, for $n \in \mathbb{N}$. We use the standard multiset extension $\prec$ of the well-founded ordering $<$ on natural numbers which is also well-founded. For each $n \in \mathbb{N}$ we define $\mathcal{W}_n(M)$ as the multiset given by the following inductive definition on M:

$$
\begin{array}{rcl}
\mathcal{W}_n(a) & \stackrel{\text{def}}{=} & \langle\!\langle\ \rangle\!\rangle \\
\mathcal{W}_n(\lambda a^A.M) & \stackrel{\text{def}}{=} & \mathcal{W}_n(M) \\
\mathcal{W}_n(M\,N) & \stackrel{\text{def}}{=} & \mathcal{W}_n(M) \uplus \mathcal{W}_n(N) \\
\mathcal{W}_n(\langle u;\sigma\rangle) & \stackrel{\text{def}}{=} & \langle\!\langle\ \rangle\!\rangle
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{W}_n(!^{\Sigma}_{e} M) & \stackrel{\text{def}}{=} & n * \mathcal{W}^t(M) \uplus \\
 & & \uplus \mathcal{W}_{n*\mathcal{W}^t(M)}(M) \\
\mathcal{W}_n(\text{LET}\, u \leftarrow M \,\text{IN}\, N) & \stackrel{\text{def}}{=} & \mathcal{W}_n(M) \uplus \mathcal{W}_n(N) \\
\mathcal{W}_n(\alpha\vartheta) & \stackrel{\text{def}}{=} & \biguplus_{i\in 1..10} \mathcal{W}_n(\vartheta(c_i)) \\
\mathcal{W}_n(e \rhd M) & \stackrel{\text{def}}{=} & \mathcal{W}_n(M)
\end{array}
$$

where $\mathcal{W}^t(M)$ is the number of free trail variables in M plus 1. Note that $\mathcal{W}^t(e \rhd M) \stackrel{\text{def}}{=} \mathcal{W}^t(M)$. The weight functions informally count the number of trail variables that are available for look-up in audited computation units. The principle reduction axiom β either erases the argument N or substitutes exactly one copy, given the affine nature of truth hypothesis. However, multiple copies of M can arise from $\beta_\Box$ reduction (cf. Fig. 5.7), possibly under "!" constructors (hence our restriction in item 2 below). Finally, we must take into account that although a trail variable is consumed by $\mathcal{I}$ it also copies the terms in ϑ (which may contain occurrences of the "!" constructor). In contrast to $\beta_\Box$ however, the consumed trail variable can be used to make the copies of "!" made by $e\vartheta$ weigh less than the outermost occurrence of "!" on the left-hand side of $\mathcal{I}$.

Lemma 5.7.7 1. $\mathcal{W}_n((\lambda a : A.M)\, N) \succeq \mathcal{W}_n(M^a_{N,t})$.

2. If M has no occurrences of the modal term constructor, then $\mathcal{W}_n(\mathsf{let}\, u : A[\Sigma] = !^{\Sigma}_{e} M \,\mathsf{in}\, N) \succ \mathcal{W}_n(\mathfrak{bb}(u^{A[\Sigma]}.t, \Sigma.s) \rhd N^u_{\Sigma.(M,s,e)})$.
3. $\mathcal{W}_n(!^{\Sigma}_{e}\mathcal{C}[\alpha\vartheta]) \succ \mathcal{W}_n(!^{\Sigma}_{e}\mathcal{C}[\mathfrak{ti}(\theta,\alpha) \rhd e\vartheta])$.

Lemma 5.7.8 Suppose M has no occurrences of the modal term constructor. Then $\mathcal{W}_n(\mathsf{let}\, u : A[\Sigma] = !^{\Sigma}_{e} M \,\mathsf{in}\, N) \succ \mathcal{W}_n(\mathfrak{bb}(u^{A[\Sigma]}.t, \Sigma.s) \rhd N^u_{\Sigma.(M,s,e)})$.

Lemma 5.7.9 $\mathcal{W}_n(!^{\Sigma}_{e}\mathcal{C}[\alpha\vartheta]) \succ \mathcal{W}_n(!^{\Sigma}_{e}\mathcal{C}[\mathfrak{ti}(\theta,\alpha) \rhd e\vartheta])$.

From these results follow:

Lemma 5.7.10 (1) $M \overset{rf}{\mapsto}_{\beta,\beta_\Box} N$ implies $\mathcal{W}_n(M) \succeq \mathcal{W}_n(N)$; (2) $M \overset{f}{\mapsto}_{\mathcal{I}} N$ implies $\mathcal{W}_n(M) \succ \mathcal{W}_n(N)$; and (3) $M \rightsquigarrow N$ implies $\mathcal{W}_n(M) = \mathcal{W}_n(N)$.

5.8 Discussion

We have presented a proof theoretical analysis of a functional computation model that keeps track of its computation history. A Curry-de Bruijn-Howard isomorphism of an affine fragment of Artemov's Justification Logic yields a lambda calculus $\lambda^{\mathfrak{h}}$ which models audited units of computation. Reduction in these units generates audit trails that are confined within them. Moreover, these units may look-up these trails and make decisions based on them. We prove type safety for $\lambda^{\mathfrak{h}}$ and strong normalisation for a restriction of it. It would be nice to lift the restriction in the proof of strong normalisation that M in the principle reduction axiom $\beta_\Box$ not have occurrences of the audited computation unit constructor "!". Also, it would make sense to study audited computation in a classical setting where, based on audit trail look-up, the current continuation could be disposed of in favour of a more judicious computation. Finally, although examples from the security domain seem promising more are needed in order to better evaluate the applicability of these ideas.

5.8.1 Related Work

We develop a proof theoretical analysis of a λ-calculus which produces a trail of its execution. This builds on ideas stemming from **JL**, a judgemental analysis of modal logic [76, 50, 52, 51] and Contextual Modal Type Theory [88]. More precisely, we argue how a fragment of **JL** whose notion of validity is relative to a context of affine variables of proof compatibility may be seen, via the Curry-de Bruijn-Howard interpretation, as a type system for a calculus that records its computation history.

S. Artemov introduced **JL** in [9, 10, 11]. For natural deduction and sequent calculus presentations consult [10, 42, 12]. Computational interpretation of proofs in **JL** are studied in [8, 12, 39], however none of these address audit trails. From a type theoretic perspective we should mention the theory of dependent types [20] where types may depend on terms, in much the same way that a type $[\![s]\!]A$ depends on the proof code s. However, dependent type theory lacks a notion of internalisation of derivations as is available in **JL**.

Finally, there is work on exploring what **JL** can contribute towards mobile code interpretations of intuitionistic S4 [79, 115, 113, 114, 66]. The result is a calculus of certified mobile units in which a unit consists of mobile code and a certificate [39]. The type system guarantees that when these mobile units are composed to form new ones, the certificates that are constructed out of those of the composed units, are correct.

Part III

Conclusions and Future Works

6 Conclusions and Future Work

6.1 Conclusions

We have addressed the issue of what is a secure bytecode program from the point of view of confidentiality in information flow. We studied two main security policies: termination insensitive non-interference and robust declassification properties.

First, we presented a type system for ensuring secure information flow in a `JVM`-like language that allows instances of a class to have fields with security levels depending on the context in which they are instantiated. This differs over extant approaches in which a global fixed security level is assigned to a field, thus improving the precision of the analysis as described in the Section 3.1. Two further contributions are that local variables are allowed to be reused with different security levels and also that we enhance precision on how information flow through the stack is handled, as described in Section 3.1.

Although noninterference is an appealing formalisation for the absence of leaking of sensitive information, its applicability in practice is somewhat limited given that many systems require intensional release of secret information. This motivated our extension of a core `JVM`-like language with a mechanism for performing downgrading of confidential information. We discussed how the downgrading mechanism may be abused in our language and then extend our type system for ensuring it captures these situations. It is proved that the type system enforces robustness of the declassification mechanism in the sense of [127]: attackers may not affect what information is released or whether information is released at all. The restriction of the attackers ability to modify data is handled by enriching the confidentiality lattice with integrity levels.

It should be mentioned that in the presence of flow-sensitive type systems we have showed that variable reuse endows the attacker with further means to affect declassification than those available in high-level languages in which variables are assigned a fixed type during their entire lifetime. Additional means of enhancing the ability of an attacker to declassify information in stack-based languages (e.g. bytecode) is by manipulating the stack, as exemplified in Section 4.1. Also, in unstructured low-level languages, we are required to restrict the use of jumps (including uncaught exceptions).

In the second and last part of this thesis, we presented a proof theoretical analysis of a functional computation model that keeps track of its computation history. A Curry-de Bruijn-Howard isomorphism of an affine fragment of Artemov's Justification Logic yields a lambda calculus $\lambda^{\natural}$ which models audited units of computation. Reduction in these units generates audit trails that are confined within them. Moreover, these units may look-up these trails and make decisions based on them. We affirmed that the computational interpretation of **JL** is a programming language that records its computation history. We proved type safety for $\lambda^{\natural}$ and strong normalisation for a restriction of it. Furthermore, we showed that $\lambda^{\natural}$ can be used for writing programs that enforce a variety of security policies based on audited trails.

6.2 Future Work

In relation to type systems for information flow and declassification:

- We are developing an extension of our type system that supports threads. Support for threads seems to be absent in the literature on IFA for bytecode languages (a recent exception being [27]) and would be welcome.
- One of the difficulties of tracking information flows is that information may flow in various indirect ways. Over 30 years ago, Lampson [71] coined the phrase covert channel to describe channels which were not intended for information transmission at all. At that time the concern was unintended transmission of information between users on timeshared mainframe computers. In much security research that followed, it was not considered worth the effort to consider covert channels. But with the increased exposure of sensitive information to potential attackers, and the ubiquitous use of cryptographic mechanisms, covert channels have emerged as a serious threat to the security of modern systems – both networked and embedded [111]. A recent work proposes techniques to deal with internal timming covert channel in multi-thread low-level languages [27, 95]. We are interested in extending the type system (or develop others) to consider covert channels, such as, certain timing leaks or resource consumption leaks.
- For increased confidence the results should be verified in the Coq Proof Assistant [46] (or similar). Furthermore, this opens the possibility of applying automatic program extraction [49] to obtain a verified OCaml implementation of the extended bytecode verifier for the JVM.
- Also, the inclusion of a primitive for *endorsement* (for upgrading the integrity of data) as studied elsewhere [85, 67] in the setting of high-level languages would be interesting. Note that although confidentiality and integrity are dual properties, the treatment of `declassify` and `endorse` should not be symmetric for otherwise the attacker would be given total control over the data. We anticipate no major difficulties in adapting *qualified robustness* [85] in our type system. Although, qualified robustness is in some cases more permissive than desired [85] and we must probe that the type system meets *posibilistic* non-interference.
- In order to validate experimentally the benefits of type system, it is important to perform case studies of real code. Potential examples include case studies developed in Jif [81], such as mental poker or battleship (in this case must be considered endorsement operation), as well as, case studies stemming from the Trusted Personal Device industry.
- To sum up, although less significant, these are worthy contributions towards minimizing the assumptions on the code that is to be analyzed hence furthering its span of applicability. In order to achieve an acceptable degree of usability, the information flow type system of relies on preliminary static analyses that provide a more accurate approximation of the control flow graph of the program. Typically, the preliminary analyses will perform safety analyses such as class analysis, null pointer analysis, exception analysis and escape analysis. For example, null pointer analysis could use to indicate whether the execution of a `putfield` instruction will be normal or there will be an exception. This analysis drastically improve the quality of the approximation of the control dependence regions.

Regarding $\mathbf{JL}^{\bullet}$and $\lambda^{\natural}$:

- It would be nice to lift the restriction in the proof of strong normalisation that M in the principle reduction axiom $\beta_{\Box}$ not have occurrences of the audited computation unit constructor "!".
- Also, it would make sense to study audited computation in a classical setting where, based on audit trail look-up, the current continuation could be disposed of in favour of a more judicious computation.
- Finally, although examples from the security domain seem promising more are needed in order to better evaluate the applicability of these ideas.

Part IV

Appendix

A

Noninterference Soundness Proofs

This appendix presents detailed proofs of all results of Chapter 3.

A.1 Preliminaries

The first five results address preservation of indistinguishability by subtyping. The proofs are either by direct inspection of the corresponding notion of indistinguishability (as, for example, in Lemma A.1) or by induction on the derivation of the indistinguishability judgement. Lemma A.7 is straightforward given that $S' \leq_l S''$, $l \not\sqsubseteq \lambda$ implies $S' = S''$.

Lemma A.1. *$\beta, l \vdash v_1 \sim^\lambda v_2$ and $\beta \subseteq \beta'$ and $l' \sqsubseteq \lambda$ implies $\beta', l' \vdash v_1 \sim^\lambda v_2$.*

Lemma A.2. *If $\beta, V, l \vdash \alpha_1 \sim^\lambda \alpha_2$, $\beta \subseteq \beta'$, $l \sqsubseteq \lambda$ and $V \leq V'$, then $\beta', V', l \vdash \alpha_1 \sim^\lambda \alpha_2$.*

Lemma A.3. *If $\beta, (V, V'), l \vdash \alpha_1 \sim^\lambda \alpha_2$, $l \not\sqsubseteq \lambda$ and $V' \leq V''$ then $\beta, (V, V''), l \vdash \alpha_1 \sim^\lambda \alpha_2$.*

Lemma A.4. *If $\beta, (V', V), l \vdash \alpha_1 \sim^\lambda \alpha_2$, $l \not\sqsubseteq \lambda$ and $V' \leq V''$ then $\beta, (V'', V), l \vdash \alpha_1 \sim^\lambda \alpha_2$.*

Proof. Immediate consequence of lemma A.1. ∎

Lemma A.5. *If $\beta, V, l \vdash \alpha_1 \sim^\lambda \alpha_2$, $l \sqsubseteq \lambda$ and $l' \sqsubseteq \lambda$ then $\beta, V, l' \vdash \alpha_1 \sim^\lambda \alpha_2$.*

Proof. We have, by hypothesis, that for all $x \in \mathbb{X} : \beta, \lfloor V(x) \rfloor \vdash \alpha_1(x) \sim^\lambda \alpha_2(x)$ holds. Let $l' \sqsubseteq \lambda$ then follows $\beta, V, l' \vdash \alpha_1 \sim^\lambda \alpha_2$ by local variable assignment indistinguishability definition. ∎

Lemma A.6. *If $\beta, S, l \vdash \sigma_1 \sim^\lambda \sigma_2$, $\beta \subseteq \beta'$, $l \sqsubseteq \lambda$ and $S \leq_l S'$ then $\beta', S', l \vdash \sigma_1 \sim^\lambda \sigma_2$.*

Lemma A.7. *If $\beta, (S, S'), l \vdash \sigma_1 \sim^\lambda \sigma_2$, $l \not\sqsubseteq \lambda$ and $S' \leq_l S''$, then $\beta, (S, S''), l \vdash \sigma_1 \sim^\lambda \sigma_2$.*

Lemma A.8. *If $\beta, (S', S), l \vdash \sigma_1 \sim^\lambda \sigma_2$, $l \not\sqsubseteq \lambda$ and $S' \leq_l S''$, then $\beta, (S'', S), l \vdash \sigma_1 \sim^\lambda \sigma_2$.*

Lemma A.9. *If $\beta, S, l \vdash \sigma_1 \sim^\lambda \sigma_2$, $l \sqsubseteq \lambda$ and $l' \sqsubseteq \lambda$ then $\beta, S, l' \vdash \sigma_1 \sim^\lambda \sigma_2$.*

Proof. We proceed by induction on the the derivation of $\beta, S, l \vdash \sigma_1 \sim^\lambda \sigma_2$. The base case is straightforward given that $\beta, \epsilon, l' \vdash \epsilon \sim^\lambda \epsilon$ follows from $l' \sqsubseteq \lambda$ and L-Nil. For the inductive case, we assume that $\beta, l'' \cdot S, l \vdash v_1 \cdot \sigma_1' \sim^\lambda v_2 \cdot \sigma_2'$ is derivable and that the derivation ends in an application of Cons-L. We must prove that $\beta, l'' \cdot S, l' \vdash v_1 \cdot \sigma_1' \sim^\lambda v_2 \cdot \sigma_2'$. By Cons-L we know that $\beta, l'' \vdash v_1 \sim^\lambda v_2$ (1) and $\beta, S, l \vdash \sigma_1 \sim^\lambda \sigma_2$. Now, by the I.H. we have that $\beta, S, l' \vdash \sigma_1 \sim^\lambda \sigma_2$ (2). Then by (1), (2) and Cons-L $\beta, l'' \cdot S, l' \vdash v_1 \cdot \sigma_1 \sim^\lambda v_2 \cdot \sigma_2$ holds. ∎

Lemma A.10. *$\beta, (T, T) \vdash \eta_1 \sim^\lambda \eta_2$ and $T \leq T'$ and $T \leq T''$ and $\beta \subseteq \beta'$ such that*

1. $\beta'_{loc} = \beta_{loc}$*,*
2. $\mathsf{Ran}(\beta'^{\triangleleft}) \subseteq \mathsf{Dom}(\eta_1)$ *and* $\mathsf{Ran}(\beta'^{\triangleright}) \subseteq \mathsf{Dom}(\eta_2)$*,*
3. $\mathsf{Dom}(\beta'^{\triangleleft}) \subseteq \mathsf{Dom}(T')$ *and* $\mathsf{Dom}(\beta'^{\triangleright}) \subseteq \mathsf{Dom}(T'')$

Then $\beta', (T', T'') \vdash \eta_1 \sim^{\lambda} \eta_2$

Proof. β' is well-defined w.r.t. η_1, η_2, T' and T'' follow directly from the β' definition. For the second item we proceed as follows. Let $o \in \mathsf{Dom}(\beta'_{loc})$ and let $f \in \mathsf{Dom}(\eta_1(o))$. We must prove:

$$\beta', \lfloor T'(\beta'^{\triangleleft -1}(o), f) \rfloor \vdash \eta_1(o, f) \sim^{\lambda} \eta_2(\beta'(o), f).$$

This result follow by hypothesis $T \leq T'$ and $T \leq T''$, i.e. $\forall a \in \mathsf{Dom}(T).\forall f \in \mathbb{F}.\lfloor T(a, f) \rfloor = \lfloor T'(a, f) \rfloor = \lfloor T''(a, f) \rfloor$. ▪

We recall from Definition 3.27 that β^{id} is a location bijection bijection set such that $\beta^{\mathsf{id}}_{loc}$ is the identity over the domain of β_{loc}.

Lemma A.11. *If* $\beta^{\mathsf{id}}, (T, T) \vdash \eta_1 \sim^{\lambda} \eta_2$ *and* $T \leq T'$*, then* $\beta^{\mathsf{id}}, (T, T') \vdash \eta_1 \sim^{\lambda} \eta_2$*.*

Proof. Immediate from the fact that for any $a \in \mathsf{Dom}(T)$ and $f \in \mathsf{Dom}(T(a))$, $\lfloor T(a, f) \rfloor = \lfloor T'(a, f) \rfloor$ and hypothesis $T \leq T'$. ▪

The following lemma shows that transitions from some high region for k to a low one through $i \mapsto i'$, implies that i' is a junction point for the latter region.

Lemma A.12. *Let* $M[\overrightarrow{\alpha}]$ *be a well-typed method with code* $B[\overrightarrow{\alpha}]$ *and* $i \in \mathsf{region}(k)$ *for some* $k \in \mathsf{Dom}(B)^{\sharp}$*. Suppose* $l \not\sqsubseteq \lambda$ *and* $\forall k' \in \mathsf{region}(k).l \sqsubseteq \mathcal{A}_{k'}$ *and furthermore let* $i \mapsto i'$*,* $\mathcal{A}_i \not\sqsubseteq \lambda$ *and* $\mathcal{A}_{i'} \sqsubseteq \lambda$*. Then* $i' = \mathsf{jun}(k)$*.*

Proof. Suppose $i' \neq \mathsf{jun}(k)$. By SOAP (property 3.10(2)) $i' \in \mathsf{region}(k)$. Furthermore, by the hypothesis $l \not\sqsubseteq \lambda$ and $\forall k' \in \mathsf{region}(k).l \sqsubseteq \mathcal{A}_{k'}$ we have $l \sqsubseteq \mathcal{A}_{i'}$. However, this contradicts $\mathcal{A}_{i'} \sqsubseteq \lambda$. ▪

A.1.1 Indistinguishability - Properties

Determining that indistinguishability of values, local variable arrays and stacks is an equivalence relation requires careful consideration. In the presence of variable reuse transitivity of indistinguishability of variable arrays in fact fails unless additional conditions are imposed. These issues are discussed in this section.

We assume that β is composable with γ in the statement of transitivity (third item) of lemmas 3.37, 3.39, A.14 and A.15 below. Furthermore, recall that we define the *inverse* of a location bijection set β, denoted $\hat{\beta}$, to be: $\hat{\beta}_{loc} = \beta^{-1}_{loc}$ and $\hat{\beta}^{\triangleleft} = \beta^{\triangleright}$ and $\hat{\beta}^{\triangleright} = \beta^{\triangleleft}$.

We can now state the first result (3.37), namely that value indistinguishability is an equivalence relation. The condition on β^{id} simply ensures that it is defined on the appropriate locations. This too is a condition that shall always be met given that in the proof of noninterference β^{id} is always taken to be the domain of the appropriate heap.

Lemma 3.37

1. $\beta^{\mathsf{id}}, l \vdash v \sim^{\lambda} v$, if $v \in \mathbb{L}$ and $l \sqsubseteq \lambda$ implies $v \in \mathsf{Dom}(\beta^{\mathsf{id}})$.
2. $\beta, l \vdash v_1 \sim^{\lambda} v_2$ implies $\hat{\beta}, l \vdash v_2 \sim^{\lambda} v_1$.
3. If $\beta, l \vdash v_1 \sim^{\lambda} v_2$ and $\gamma, l \vdash v_2 \sim^{\lambda} v_3$, then $\gamma \circ \beta, l \vdash v_1 \sim^{\lambda} v_3$.

Proof. The first two items follow directly from a close inspection of the definition of value indistinguishability. Regarding transitivity we consider the two cases, $l \not\sqsubseteq \lambda$ and $l \sqsubseteq \lambda$. In the former case, $\gamma \circ \beta, l \vdash v_1 \sim^\lambda v_3$ holds trivially by definition of value indistinguishability. For the latter, if $v_1 = \mathit{null}$ or $v_1 \in \mathbb{Z}$ we resort to transitivity of equality. If $v_1 \in \mathbb{L}$, then by hypothesis and by the definition of value indistinguishability, $v_2, v_3 \in \mathbb{L}$, $\beta(v_1) = v_2$ and $\gamma(v_2) = v_3$. And by definition of $\gamma \circ \beta$, $(\gamma \circ \beta)(v_1) = \gamma(\beta(v_1)) = v_3$. Hence $\gamma \circ \beta, l \vdash v_1 \sim^\lambda v_3$. ▪

Also, the following notions will be required for properly stating the conditions under which indistinguishability of frames, stacks, heaps and states are equivalence relations.

Notation A.13

- $\mathsf{LowLoc}(\alpha, V, \lambda)$ (or simply $\mathsf{LowLoc}(\alpha)$ if the V is understood from the context) is shorthand for $\{o \,|\, \alpha(x) = o, V(x) \sqsubseteq \lambda\}$.
- $\mathsf{LowLoc}(\sigma, S, \lambda)$ (or simply $\mathsf{LowLoc}(\sigma)$ if the S is understood from the context) is defined as $\{o \,|\, \exists i \in 0..\|S\| - 1.\sigma(i) = o, S(i) \sqsubseteq \lambda\}$
- $\mathsf{LowLoc}(\eta, \beta, T, \lambda)$ (or simply $\mathsf{LowLoc}(\eta, \beta)$ if the T is understood from the context) is defined as $\{o' \,|\, o \in \mathsf{Dom}(\beta), \exists f \in \mathsf{Dom}(\eta(o)).(\eta(o, f) = o', T(\beta^{-1}(o), f) \sqsubseteq \lambda\}$
- $\mathsf{LowLoc}(\langle i, \alpha, \sigma, \eta\rangle, \beta, \langle i, V, S, T\rangle, \lambda)$ (or simply $\mathsf{LowLoc}(s, \beta, \lambda)$ if the $\langle i, V, S, T\rangle$ is understood from the context) is defined as $\mathsf{LowLoc}(\alpha, V, \lambda) \cup \mathsf{LowLoc}(\sigma, S, \lambda) \cup \mathsf{LowLoc}(\eta, \beta, T, \lambda)$

We now address local variable arrays. Variable reuse allows a public variable to be reused for storing secret information in a high security execution context. Suppose, therefore, that $l \not\sqsubseteq \lambda$, $\beta, (V_1, V_2), l \vdash \alpha_1 \sim^\lambda \alpha_2$ and $\gamma, (V_2, V_3), l \vdash \alpha_2 \sim^\lambda \alpha_3$ where, for some x, $V_1(x) = \mathtt{L}$, $V_2(x) = \mathtt{H}$ and $V_3(x) = \mathtt{L}$. Clearly it is not necessarily the case that $\gamma \circ \beta, (V_1, V_3), l \vdash \alpha_1 \sim^\lambda \alpha_3$ given that $\alpha_1(x)$ and $\alpha_3(x)$ may differ. We thus require that either V_1 or V_3 have at least the level of V_2 for this variable: $V_2(x) \sqsubseteq V_1(x)$ or $V_2(x) \sqsubseteq V_3(x)$. Of course, it remains to be seen that such a condition can be met when proving noninterference. For now we state the result, namely that indistinguishability of local variable arrays is an equivalence relation (its proof is an easy consequence of lemma 3.37).

Lemma 3.39 Frame indistinguishability is an equivalence relation.

For low indistinguishability we have ($l \sqsubseteq \lambda$):

1. $\beta^{\mathsf{id}}, V, l \vdash \alpha \sim^\lambda \alpha$, if $\mathsf{LowLoc}(\alpha, V, \lambda) \subseteq \mathsf{Dom}(\beta^{\mathsf{id}})$.
2. $\beta, V, l \vdash \alpha_1 \sim^\lambda \alpha_2$ implies $\hat{\beta}, V, l \vdash \alpha_2 \sim^\lambda \alpha_1$.
3. If $\beta, V, l \vdash \alpha_1 \sim^\lambda \alpha_2$, $\gamma, V, l \vdash \alpha_2 \sim^\lambda \alpha_3$, then $\gamma \circ \beta, V, l \vdash \alpha_1 \sim^\lambda \alpha_3$.

For high indistinguishability ($l \not\sqsubseteq \lambda$) we have:

1. $\beta^{\mathsf{id}}, (V, V), l \vdash \alpha \sim^\lambda \alpha$, if $\mathsf{LowLoc}(\alpha, V, \lambda) \subseteq \mathsf{Dom}(\beta^{\mathsf{id}})$.
2. $\beta, (V_1, V_2), l \vdash \alpha_1 \sim^\lambda \alpha_2$ implies $\hat{\beta}, (V_2, V_1), l \vdash \alpha_2 \sim^\lambda \alpha_1$.
3. If $\beta, (V_1, V_2), l \vdash \alpha_1 \sim^\lambda \alpha_2$, $\gamma, (V_2, V_3), l \vdash \alpha_2 \sim^\lambda \alpha_3$ and $\forall x \in \mathbb{X}.V_2(x) \sqsubseteq V_1(x)$ or $V_2(x) \sqsubseteq V_3(x)$, then $\gamma \circ \beta, (V_1, V_3), l \vdash \alpha_1 \sim^\lambda \alpha_3$.

Proof. First we address the low case. Reflexivity and symmetry are immediate consequences of Lemma 3.37. We develop the case for transitivity. By hypothesis the following statement holds

$$\forall x \in \mathbb{X}.\beta, \lfloor V(x) \rfloor \vdash \alpha_1(x) \sim^\lambda \alpha_2(x) \text{ and } \forall x \in \mathbb{X}.\gamma, \lfloor V(x) \rfloor \vdash \alpha_2(x) \sim^\lambda \alpha_3(x) \tag{A.1}$$

Let $x \in \mathbb{X}$. If $V(x) \not\sqsubseteq \lambda$, then we are done. If $V(x) \sqsubseteq \lambda$, from (A.1),

$$\beta, \lfloor V(x) \rfloor \vdash \alpha_1(x) \sim^\lambda \alpha_2(x) \text{ and } \gamma, \lfloor V(x) \rfloor \vdash \alpha_2(x) \sim^\lambda \alpha_3(x) \tag{A.2}$$

and we obtain the desired result from (A.2) and Lemma 3.37.

We now address the high case. Since reflexivity and symmetry are straightforward we concentrate on transitivity. By hypothesis the following two statements hold

$$\forall x \in \mathbb{X}.\beta, \lfloor V_1(x) \sqcup V_2(x) \rfloor \vdash \alpha_1(x) \sim^\lambda \alpha_2(x) \text{ and } \forall x \in \mathbb{X}.\gamma, \lfloor V_2(x) \sqcup V_3(x) \rfloor \vdash \alpha_2(x) \sim^\lambda \alpha_3(x) \tag{A.3}$$

and

$$\forall x \in \mathbb{X}.V_2(x) \sqsubseteq V_1(x) \text{ or } V_2(x) \sqsubseteq V_3(x) \tag{A.4}$$

We must verify that $\forall x \in \mathbb{X}.\gamma \circ \beta, \lfloor V_1(x) \sqcup V_3(x) \rfloor \vdash \alpha_1(x) \sim^\lambda \alpha_3(x)$.

Let $x \in \mathbb{X}$. If $V_1(x) \sqcup V_3(x) \not\sqsubseteq \lambda$, then we are done. Otherwise, $V_1(x) \sqsubseteq \lambda$ and $V_3(x) \sqsubseteq \lambda$. From (A.4) we deduce that $V_2(x) \sqsubseteq \lambda$. Then by (A.3),

$$\beta, l \vdash \alpha_1(x) \sim^\lambda \alpha_2(x) \text{ and } \gamma, \lfloor l \rfloor \vdash \alpha_2(x) \sim^\lambda \alpha_3(x), \quad l \sqsubseteq \lambda \tag{A.5}$$

Finally, we obtain the desired result from (A.5) and Lemma 3.37.

∎

The case of stacks are dealt with similarly. First a word on notation. If S is a a stack type, we write S^n for the prefix of S of length n and S^{-n} for the suffix of S of length n. By example, let the stack type $S = v_1 \cdot v_2 \cdot v_3 \cdot v_4 \cdot v_5$, the prefix $S^3 = v_1 \cdot v_2 \cdot v_3$ and the suffix $S^{-3} = v_3 \cdot v_4 \cdot v_5$

Lemma A.14. *The stack indistinguishability relation is an equivalence relation.*

1. $\beta^{\mathsf{id}}, (S, S), l \vdash \sigma \sim^\lambda \sigma$, *if* $\mathit{LowLoc}(\alpha, S, \lambda) \subseteq \mathsf{Dom}(\beta^{\mathsf{id}}_{loc})$.
2. $\beta, (S_1, S_2), l \vdash \sigma_1 \sim^\lambda \sigma_2$ *implies* $\hat{\beta}, (S_2, S_1), l \vdash \sigma_2 \sim^\lambda \sigma_1$.
3. *If* $\beta, (S_1, S_2), l \vdash \sigma_1 \sim^\lambda \sigma_2$ *and* $\gamma, (S_2, S_3), l \vdash \sigma_2 \sim^\lambda \sigma_3$, *then* $\gamma \circ \beta, (S_1, S_3), l \vdash \sigma_1 \sim^\lambda \sigma_3$.

Proof. The low case is proved straightforwardly. We consider the high case. Reflexivity follows from reflexivity for the low case given that

$$\frac{\beta^{\mathsf{id}}, S, l \vdash \sigma \sim^\lambda \sigma' \quad l \sqsubseteq \lambda \quad l' \not\sqsubseteq \lambda}{\beta^{\mathsf{id}}, (S, S), l' \vdash \sigma \sim^\lambda \sigma'} \text{ H-Low}$$

For symmetry we proceed by structural induction on the derivation of $\beta, (S_1, S_2), l \vdash \sigma_1 \sim^\lambda \sigma_2$, with $l \not\sqsubseteq \lambda$. For the base case we resort to symmetry for the low case. The two remaining inductive cases are developed below. We assume that

$$\beta, (S_1, S_2), l \vdash \sigma_1 \sim^\lambda \sigma_2 \text{ implies } \hat{\beta}, (S_2, S_1), l \vdash \sigma_2 \sim^\lambda \sigma_1 \tag{A.6}$$

1. if $\beta, (l' \cdot S_1, S_2), l \vdash v \cdot \sigma_1 \sim^\lambda \sigma_2$, $l' \not\sqsubseteq \lambda$ by H-Cons-L, then $\beta, (S_1, S_2), l \vdash \sigma_1 \sim^\lambda \sigma_2$ and by (A.6) we can assert $\hat{\beta}, (S_2, S_1), l \vdash \sigma_2 \sim^\lambda \sigma_1$. Now by H-Cons-R we have $\hat{\beta}, (S_2, l' \cdot S_1), l \vdash \sigma_2 \sim^\lambda v \cdot \sigma_1$.
2. if $\beta, (S_1, l' \cdot S_2), l \vdash \sigma_1 \sim^\lambda v \cdot \sigma_2$, $l' \not\sqsubseteq \lambda$ by H-Cons-R, then from (A.6) and H-Cons-L we deduce $\hat{\beta}, (l' \cdot S_2, S_1), l \vdash v \cdot \sigma_2 \sim^\lambda \sigma_1$.

Transitivity may be proved by structural induction on the derivation of $\beta, (S_1, S_2), l \vdash \sigma_1 \sim^\lambda \sigma_2$ and then performing case analysis on the form of the derivation for $\gamma, (S_2, S_3), l \vdash \sigma_2 \sim^\lambda \sigma_3$.

∎

Lemma A.15. *Heap indistinguishability is an equivalence relation.*

1. $\beta^{\mathsf{id}}, (T, T) \vdash \eta \sim^\lambda \eta$, *where* $\mathsf{Dom}(\beta^{\mathsf{id}}_{loc}) \subseteq \mathsf{Dom}(\eta)$, $\mathsf{Dom}(\beta^{\mathsf{id}\triangleleft}), \mathsf{Dom}(\beta^{\mathsf{id}\triangleright}) \subseteq \mathsf{Dom}(T)$ *and* $\mathit{LowLoc}(\eta, \beta^{\mathsf{id}}, T, \lambda) \subseteq \mathsf{Dom}(\beta^{\mathsf{id}}_{loc})$.
2. $\beta, (T_1, T_2) \vdash \eta_1 \sim^\lambda \eta_2$ *implies* $\hat{\beta}, (T_2, T_1) \vdash \eta_2 \sim^\lambda \eta_1$ *assuming for every* $o \in \mathsf{Dom}(\beta)$, $f \in \eta_1(o, f)$, $T_1(\beta^{\triangleleft -1}(o), f) = T_2(\beta^{\triangleright -1}(\beta(o)), f)$.
3. *If* $\beta, (T_1, T_2) \vdash \eta_1 \sim^\lambda \eta_2$, $\gamma, (T_2, T_3) \vdash \eta_2 \sim^\lambda \eta_3$ *then* $\gamma \circ \beta, (T_1, T_3) \vdash \eta_1 \sim^\lambda \eta_3$.

Proof. Reflexivity follows directly from hypothesis and the definition of β^{id}. For symmetry we see that the four first items of high-indistinguishable heap definition follow directly from hypothesis $\beta, (T_1, T_2) \vdash \eta_1 \sim^\lambda \eta_2$ and the definition of $\hat{\beta}$. Now, we address the last item. We must verify that $\forall o' \in \mathsf{Dom}(\hat{\beta})$ and $\forall f \in \mathsf{Dom}(\eta_2(o'))$

$$\hat{\beta}, T_2(\hat{\beta}^{\lhd-1}(o'), f) \vdash \eta_2(o', f) \sim^\lambda \eta_1(\hat{\beta}(o'), f) \tag{A.7}$$

By hypothesis, $\forall o \in \mathsf{Dom}(\beta)$ and $\forall f \in \mathsf{Dom}(\eta_1(o))$

$$\beta, T_1(\beta^{\lhd-1}(o, f)) \vdash \eta_1(o, f) \sim^\lambda \eta_2(\beta(o), f) \tag{A.8}$$

By definition of $\hat{\beta}$, if $\beta(o) = o'$ then $\hat{\beta}(o') = o$. Furthermore, $\hat{\beta}^{\lhd}_{loc} = \beta^{\rhd}_{loc}$ and $\hat{\beta}^{\rhd}_{loc} = \beta^{\lhd}_{loc}$. Then we can rewrite the above statement as

$$\begin{gathered}\forall o' \in \mathsf{Dom}(\hat{\beta}).\forall f \in \mathsf{Dom}(\eta_2(o')).\\ \hat{\beta}, T_1(\hat{\beta}^{\rhd-1}(\hat{\beta}(o'), f) \vdash \eta_1(\hat{\beta}(o'), f) \sim^\lambda \eta_2(o', f)\end{gathered} \tag{A.9}$$

Finally, from the assumption $T_1(\beta^{\lhd-1}(o), f) = T_2(\beta^{\rhd-1}(\beta(o)), f)$ and the definition of $\hat{\beta}$

$$\begin{gathered}\forall o' \in \mathsf{Dom}(\hat{\beta}).\forall f \in \mathsf{Dom}(\eta_2(o')).\\ \hat{\beta}, T_2(\hat{\beta}^{\lhd-1}(\hat{\beta}(o'), f) \vdash \eta_1(\hat{\beta}(o'), f) \sim^\lambda \eta_2(o', f)\end{gathered} \tag{A.10}$$

We obtain the desired result from (A.10) and Lemma 3.37

Finally, we address transitivity. By hypothesis the following statements hold

$$\begin{gathered}\mathsf{Dom}(\beta) \subseteq \mathsf{Dom}(\eta_1), \mathsf{Ran}(\beta) \subseteq \mathsf{Dom}(\eta_2)\\ \text{and}\\ \mathsf{Dom}(\gamma) \subseteq \mathsf{Dom}(\eta_2), \mathsf{Ran}(\gamma) \subseteq \mathsf{Dom}(\eta_3)\end{gathered} \tag{A.11}$$

$$\begin{gathered}\mathsf{Ran}(\beta^{\lhd}_{loc}) \subseteq \mathsf{Dom}(\eta_1), \mathsf{Ran}(\beta^{\rhd}_{loc}) \subseteq \mathsf{Dom}(\eta_2)\\ \text{and}\\ \mathsf{Ran}(\gamma^{\lhd}_{loc}) \subseteq \mathsf{Dom}(\eta_2), \mathsf{Ran}(\gamma^{\rhd}_{loc}) \subseteq \mathsf{Dom}(\eta_3)\end{gathered} \tag{A.12}$$

$$\begin{gathered}\mathsf{Dom}(\beta^{\lhd}_{loc}) \subseteq \mathsf{Dom}(T_1), \mathsf{Dom}(\beta^{\rhd}_{loc}) \subseteq \mathsf{Dom}(T_2)\\ \text{and}\\ \mathsf{Dom}(\gamma^{\lhd}_{loc}) \subseteq \mathsf{Dom}(T_2), \mathsf{Dom}(\gamma^{\rhd}_{loc}) \subseteq \mathsf{Dom}(T_3)\end{gathered} \tag{A.13}$$

$$\begin{gathered}\forall o \in \mathsf{Dom}(\beta).\mathsf{Dom}(\eta_1(o)) = \mathsf{Dom}(\eta_2(\beta(o)))\\ \text{and}\\ \forall o' \in \mathsf{Dom}(\gamma).\mathsf{Dom}(\eta_2(o')) = \mathsf{Dom}(\eta_3(\gamma(o')))\end{gathered} \tag{A.14}$$

We prove the first item of high-indistinguishable heap definition directly from (A.11), (A.12), (A.14) and (A.13) and the definition of $\gamma \circ \beta$.

We now address the second item. By hypothesis we know that

$$\begin{gathered}\forall o \in \mathsf{Dom}(\beta).\forall f \in \mathsf{Dom}(\eta_1(o)).\\ \beta, T_1(\beta^{\lhd-1}(o), f) \vdash \eta_1(o, f) \sim^\lambda \eta_2(\beta(o), f)\end{gathered} \tag{A.15}$$

and

$$\begin{gathered}\forall o' \in \mathsf{Dom}(\gamma).\forall f \in \mathsf{Dom}(\eta_2(o')).\\ \gamma, T_2(\gamma^{\lhd-1}(o'), f) \vdash \eta_2(o', f) \sim^\lambda \eta_3(\gamma(o'), f)\end{gathered} \tag{A.16}$$

Take any $o \in \mathsf{Dom}(\gamma \circ \beta)$. Thus $o \in \mathsf{Dom}(\beta)$ and $\beta(o) \in \mathsf{Dom}(\gamma)$, given that β is composable with γ. Then by (A.15) and (A.16) we have

$$\begin{gathered}\forall o \in \mathsf{Dom}(\beta).\forall f \in \mathsf{Dom}(\eta_1(o)).\\ \beta, T_1(\beta^{\lhd-1}(o), f) \sqcup T_2(\beta^{\rhd-1}(\beta(o)), f) \vdash \eta_1(o, f) \sim^\lambda \eta_2(\beta(o), f)\end{gathered} \tag{A.17}$$

and

$$\begin{array}{c}\forall \beta(o) \in \mathsf{Dom}(\gamma).\forall f \in \mathsf{Dom}(\eta_2(\beta(o))). \\ \gamma, T_2(\gamma^{\lhd -1}(\beta(o)), f) \vdash \eta_2(\beta(o), f) \sim^{\lambda} \eta_3(\gamma(\beta(o)), f)\end{array} \tag{A.18}$$

Since $T_1(\beta^{\lhd -1}(o), f) = T_2(\beta^{\rhd -1}(\beta(o)), f)$ and for all $o \in \mathsf{Dom}(\beta)$, $\beta_{loc}^{\rhd -1}(\beta(o)) = \gamma^{\lhd -1}(\beta(o))$ (β is composable with γ),

$$T_1(\beta^{\lhd -1}(o), f) = T_2(\gamma^{\lhd -1}(\beta(o)), f)$$

Therefore we resort to transitivity of indistinguishability of values (Lemma 3.37) to conclude that

$$\forall o \in \mathsf{Dom}(\beta).\forall f \in \mathsf{Dom}(\eta_1(o)).\beta, T_1(\beta^{\lhd -1}(o), f) \vdash \eta_1(o, f) \sim^{\lambda} \eta_3(\gamma(\beta(o)), f)$$

Finally, note that

$$\forall o \in \mathsf{Dom}(\gamma \circ \beta), \forall f \in \mathsf{Dom}(\eta_1(o)).T_1(\beta^{\lhd -1}(o), f) = T_3(\gamma^{\lhd -1}(\gamma(\beta(o))), f)$$

∎

Then, the proof of Lemma3.40, machine state indistinguishability is an equivalence relation, follow by the before results.

A.2 Low Context

We now address the proof of the Equivalence in Low Level Contexts.

Lemma 3.44 (One-Step Low)
Suppose

1. $\mathcal{A}_i \sqsubseteq \lambda$;
2. $\mathtt{pc}(s_1) = \mathtt{pc}(s_2)$
3. $s_1 \longrightarrow_B s_1'$;
4. $s_2 \longrightarrow_B s_2'$; and
5. $\beta \vdash s_1 \sim^{\lambda} s_2$ for some location bijection set β.

Then
$\beta' \vdash s_1' \sim^{\lambda} s_2'$ and $\mathtt{pc}(s_1') = \mathtt{pc}(s_2')$, for some $\beta' \supseteq \beta$.

Proof. We proceed by case analysis on the instruction that is executed.

Case: Suppose $B(i) = \mathtt{store}\ x$. Then

$$\frac{B(i) = \mathtt{store}\ x}{\langle i, \alpha_1, v_1 \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i+1, \alpha_1 \oplus \{x \mapsto v_1\}, \sigma_1, \eta_1 \rangle = s_1'}$$

$$\frac{B(i) = \mathtt{store}\ x}{\langle i, \alpha_2, v_2 \cdot \sigma_2, \eta_2 \rangle \longrightarrow_B \langle i+1, \alpha_2 \oplus \{x \mapsto v_2\}, \sigma_2, \eta_2 \rangle = s_2'}$$

Moreover, by T-Store:

$$\begin{array}{l}\mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{V}_{i+1}(x) \cdot \mathcal{S}_{i+1} \\ \mathcal{V}_i \setminus x \leq \mathcal{V}_{i+1} \setminus x \\ \mathcal{A}_i \sqsubseteq \mathcal{V}_{i+1}(x) \\ \mathcal{T}_i \leq \mathcal{T}_{i+1}\end{array}$$

We prove $\beta \vdash s'_1 \sim^\lambda s'_2$.

1. Since $i'_1 = i+1 = i'_2$, then $\mathcal{A}_{i'_1} = \mathcal{A}_{i'_2}$.
2. First we note that $\beta, \lfloor \mathcal{V}_{i+1}(x) \rfloor \vdash v_1 \sim^\lambda v_2$ **(1)** follows from hypothesis $\beta, \lfloor \mathcal{S}_i(0) \rfloor \vdash v_1 \sim^\lambda v_2$, condition 1 of T-Store ($\mathcal{S}_i(0) \sqsubseteq \mathcal{V}_{i+1}(x)$) and Lemma A.1 .
Furthermore, by hypothesis $\beta, \mathcal{V}_i \setminus x, \lfloor \mathcal{A}_i \rfloor \vdash \alpha_1 \setminus x \sim^\lambda \alpha_2 \setminus x, \mathcal{A}_i \sqsubseteq \lambda$ and by condition 2 of T-Store and Lemma A.2 we have $\beta, \mathcal{V}_{i+1} \setminus x, \mathcal{A}_i \vdash \alpha_1 \setminus x \sim^\lambda \alpha_2 \setminus x$ **(2)**.
From **(1)** and **(2)** we deduce $\beta, \mathcal{V}_{i+1}, \mathcal{A}_i \vdash \alpha_1 \sim^\lambda \alpha_2$ **(3)**.
If $\mathcal{A}_{i+1} \sqsubseteq \lambda$ then $\beta, \mathcal{V}_{i+1}, \mathcal{A}_{i+1} \vdash \alpha_1 \sim^\lambda \alpha_2$ follows by **(3)** and Lemma A.5.
If $\mathcal{A}_{i+1} \not\sqsubseteq \lambda$, then we note that $\beta, \mathcal{V}_{i+1}, \mathcal{A}_{i+1} \vdash \alpha_1 \sim^\lambda \alpha_2$ follows from **(3)**.
3. By hypothesis $\beta, \mathcal{S}_i, \mathcal{A}_i \vdash v_1 \cdot \sigma_1 \sim^\lambda v_2 \cdot \sigma_2$, condition 1 of T-Store and lemma A.6 we have $\beta, \mathcal{V}_{i+1}(x) \cdot \mathcal{S}_{i+1}, \mathcal{A}_i \vdash v_1 \cdot \sigma_1 \sim^\lambda v_2 \cdot \sigma_2$.
Now, by $\mathcal{A}_i \sqsubseteq \lambda$ and Cons-L we have $\beta, \mathcal{S}_{i+1}, \mathcal{A}_i \vdash \sigma_1 \sim^\lambda \sigma_2$.
We have two cases:
   - $\mathcal{A}_{i+1} \sqsubseteq \lambda$. Then by Lem. A.9 we have $\beta, \mathcal{S}_{i+1}, \mathcal{A}_{i+1} \vdash \sigma_1 \sim^\lambda \sigma_2$, as required.
   - $\mathcal{A}_{i+1} \not\sqsubseteq \lambda$. Then by H-Low $\beta, (\mathcal{S}_{i+1}, \mathcal{S}_{i+1}), \mathcal{A}_{i+1} \vdash \sigma_1 \sim^\lambda \sigma_2$ holds.
4. By hypothesis $\beta, \mathcal{T}_i \vdash \eta_1 \sim^\lambda \eta_2$, condition 4 of `T-Store` and Lemma A.10, $\beta, \mathcal{T}_{i+1} \vdash \eta_1 \sim^\lambda \eta_2$.

Case: $B(i) = \texttt{load}\ x$. By the operational semantics:

$$\frac{B(i) = \texttt{load}\ x}{\langle i, \alpha_1, \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i+1, \alpha_1, \alpha_1(x) \cdot \sigma_1, \eta_1 \rangle = s'_1}$$

$$\frac{B(i) = \texttt{load}\ x}{\langle i, \alpha_2, \sigma_2, \eta_2 \rangle \longrightarrow_B \langle i+1, \alpha_2, \alpha_2(x) \cdot \sigma_2, \eta_2 \rangle = s'_2}$$

Moreover, by T-Load:

$$\begin{array}{l} \mathcal{A}_i \sqcup \mathcal{V}_i(x) \cdot \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \end{array}$$

We prove $\beta \vdash s'_1 \sim^\lambda s'_2$.

1. By the operational semantics $i'_1 = i+1 = i'_2$, hence $\mathcal{A}_{i'_1} = \mathcal{A}_{i'_1}$.
2. We must prove $\beta, \mathcal{V}_{i+1}, \lfloor \mathcal{A}_{i+1} \rfloor \vdash \alpha_1 \sim^\lambda \alpha_2$. Given that $\mathcal{A}_{i+1}$ may be either low or high we must consider both cases. Therefore, suppose first that $\mathcal{A}_{i+1} \sqsubseteq \lambda$. By hypothesis we know $\beta, \mathcal{V}_i, \lfloor \mathcal{A}_i \rfloor \vdash \alpha_1 \sim^\lambda \alpha_2$, $\mathcal{A}_i \sqsubseteq \lambda$. Then, resorting to condition 2 of T-Load and Lemma A.2, $\beta, \mathcal{V}_{i+1}, \lfloor \mathcal{A}_i \rfloor \vdash \alpha_1 \sim^\lambda \alpha_2$ Now, by Lemma A.5 the result follows.
Suppose now that $\mathcal{A}_{i+1} \not\sqsubseteq \lambda$. We proceed as above and simply note that by the definition of indistinguishability of local variable frames $\beta, \mathcal{V}_{i+1}, \lfloor \mathcal{A}_{i+1} \rfloor \vdash \alpha_1 \sim^\lambda \alpha_2$ follows from $\beta, \mathcal{V}_{i+1}, \lfloor \mathcal{A}_i \rfloor \vdash \alpha_1 \sim^\lambda \alpha_2$.
3. We have two cases:
   - $\mathcal{A}_{i+1} \sqsubseteq \lambda$. By hypothesis we know $\beta, \mathcal{S}_i, \mathcal{A}_i \vdash \sigma_1 \sim^\lambda \sigma_2$, $\mathcal{A}_i \sqsubseteq \lambda$ and $\beta, \lfloor \mathcal{V}_i(x) \rfloor \vdash \alpha_1(x) \sim^\lambda \alpha_2(x)$. From the latter and Lemma A.1 we deduce $\beta, \lfloor \mathcal{A}_i \sqcup \mathcal{V}_i(x) \rfloor \vdash \alpha_1(x) \sim^\lambda \alpha_2(x)$ Then, by L-Cons,
$$\beta, \mathcal{A}_i \sqcup \mathcal{V}_i(x) \cdot \mathcal{S}_i, \mathcal{A}_i \vdash \alpha_1(x) \cdot \sigma_1 \sim^\lambda \alpha_2(x) \cdot \sigma_2 \tag{A.19}$$
Condition 1 of T-Load, namely $\mathcal{A}_i \sqcup \mathcal{V}_i(x) \cdot \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{i+1}$, and (A.19) allows us to apply Lemma A.6 to deduce $\beta, \mathcal{S}_{i+1}, \mathcal{A}_i \vdash \alpha_1(x) \cdot \sigma_1 \sim^\lambda \alpha_2(x) \cdot \sigma_2$. Now, by Lem. A.9 follows the result.
   - $\mathcal{A}_{i+1} \not\sqsubseteq \lambda$. By the previous case we know $\beta, \mathcal{S}_{i+1}, \mathcal{A}_i \vdash \alpha_1(x) \cdot \sigma_1 \sim^\lambda \alpha_2(x) \cdot \sigma_2$. Then by H-Low $\beta, (\mathcal{S}_{i+1}, \mathcal{S}_{i+1}), \mathcal{A}_{i+1} \vdash \alpha_1(x) \cdot \sigma_1 \sim^\lambda \alpha_2(x) \cdot \sigma_2$ holds.
4. By hypothesis $\beta, \mathcal{T}_i \vdash \eta_1 \sim^\lambda \eta_2$, condition 3 of `T-Load` and Lemma A.10, $\beta, \mathcal{T}_{i+1} \vdash \eta_1 \sim^\lambda \eta_2$.

Case: $B(i) = \texttt{goto}\ i'$
By the operational semantics we know:

$$\frac{B(i) = \texttt{goto}\ i'}{\langle i, \alpha_1, \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i', \alpha_1, \sigma_1, \eta_1 \rangle = s'_1}$$

$$\frac{B(i) = \texttt{goto}\ i'}{\langle i, \alpha_2, \sigma_2, \eta_2 \rangle \longrightarrow_B \langle i', \alpha_2, \sigma_2, \eta_2 \rangle = s'_2}$$

By the T-Goto rule we know:

$$\begin{array}{l} \mathcal{V}_i \leq \mathcal{V}_{i'} \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\ \mathcal{T}_i \leq \mathcal{T}_{i'} \end{array}$$

We set $\beta' = \beta$ and prove $\beta \vdash s'_1 \sim^\lambda s'_2$.

1. By the operational semantics $i'_1 = i' = i'_2$ and hence $\mathcal{A}_{i'_1} = \mathcal{A}_{i'_1}$.
2. Suppose first that $\mathcal{A}_{i'} \sqsubseteq \lambda$. By hypothesis $\beta, \mathcal{V}_i, \mathcal{A}_i \vdash \alpha_1 \sim^\lambda \alpha_2$, condition 1 of T-Goto and Lemma A.2, $\beta, \mathcal{V}'_i, \mathcal{A}_i \vdash \alpha_1 \sim^\lambda \alpha_2$. The result follows by Lemma A.5.
   If $\mathcal{A}_{i'} \not\sqsubseteq \lambda$, we proceed as above and simply note that $\beta, \mathcal{V}'_i, \mathcal{A}_{i'} \vdash \alpha_1 \sim^\lambda \alpha_2$ follows from $\beta, \mathcal{V}'_i, \mathcal{A}_i \vdash \alpha_1 \sim^\lambda \alpha_2$.
3. Suppose first that $\mathcal{A}_{i'} \sqsubseteq \lambda$. By hypothesis $\beta, \mathcal{S}_i, \mathcal{A}_i \vdash \sigma_1 \sim^\lambda \sigma_2$. and by condition 2 of T-Goto and Lemma A.6 $\beta, \mathcal{S}_{i'}, \mathcal{A}_i \vdash \sigma_1 \sim^\lambda \sigma_2$. Now, by Lem. A.9 follows the result.
   If $\mathcal{A}_{i'} \not\sqsubseteq \lambda$, we proceed as above and note that $\beta, \mathcal{S}_{i'}, \mathcal{A}_{i'} \vdash \sigma_1 \sim^\lambda \sigma_2$ follows from $\beta, \mathcal{S}_{i'}, \mathcal{A}_i \vdash \sigma_1 \sim^\lambda \sigma_2$ by resorting to H-Low.
4. By hypothesis $\beta, \mathcal{T}_i \vdash \eta_1 \sim^\lambda \eta_2$, condition 3 of T-Goto and Lemma A.10, $\beta, \mathcal{T}_{i'} \vdash \eta_1 \sim^\lambda \eta_2$.

Case: $B(i) = \texttt{if}\ i'$
We must consider four cases. In all of them we take $\beta' = \beta$:

1. By the operational semantics we know:

$$\frac{B(i) = \texttt{if}\ i' \quad v_1 \neq 0}{\langle i, \alpha_1, v_1 \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i+1, \alpha_1, \sigma_1, \eta_1 \rangle = s'_1}$$

$$\frac{B(i) = \texttt{if}\ i' \quad v_1 \neq 0}{\langle i, \alpha_2, v_2 \cdot \sigma_2, \eta_2 \rangle \longrightarrow_B \langle i+1, \alpha_2, \sigma_2, \eta_2 \rangle = s'_2}$$

   From T-If:

$$\begin{array}{l} \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \kappa \cdot \mathcal{S}_{i+1} \\ \mathcal{A}_i \sqsubseteq \kappa \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \\ \forall k \in \mathsf{region}(i). \kappa \sqsubseteq \mathcal{A}_k \end{array}$$

   We prove $\beta \vdash s'_1 \sim^\lambda s'_2$.
   a) By the operational semantics $i'_1 = i+1 = i'_2$, hence $\mathcal{A}_{i'_1} = \mathcal{A}_{i'_1}$.

b) If $\mathcal{A}_{i+1} \sqsubseteq \lambda$. The $\beta, \mathcal{V}_i, \mathcal{A}_i \vdash \alpha_1 \sim^\lambda \alpha_2$, condition 1 of T-If and Lemma A.2, we have $\beta, \mathcal{V}_{i+1}, \mathcal{A}_i \vdash \alpha_1 \sim^\lambda \alpha_2$. Now, the result follows from apply Lemma A.5.
If $\mathcal{A}_{i+1} \not\sqsubseteq \lambda$. $\beta, \mathcal{V}_{i+1}, \mathcal{A}_{i+1} \vdash \alpha_1 \sim^\lambda \alpha_2$ follows from $\beta, \mathcal{V}_{i+1}, \mathcal{A}_i \vdash \alpha_1 \sim^\lambda \alpha_2$.
c) Idem to T-Goto case.
d) Idem to T-Goto case.

2. By the operational semantics we know:

$$\frac{B(i) = \texttt{if } i' \quad v_1 = 0}{\langle i, \alpha_1, v_1 \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i', \alpha_1, \sigma_1, \eta_1 \rangle = s_1'}$$

$$\frac{B(i) = \texttt{if } i' \quad v_1 = 0}{\langle i, \alpha_2, v_2 \cdot \sigma_2, \eta_2 \rangle \longrightarrow_B \langle i', \alpha_2, \sigma_2, \eta_2 \rangle = s_2'}$$

From T-If:

$$\begin{array}{l} \mathcal{V}_i \leq \mathcal{V}_{i'} \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \kappa \cdot \mathcal{S}_{i'} \\ \mathcal{A}_i \sqsubseteq \kappa \\ \mathcal{T}_i \leq \mathcal{T}_{i'} \\ \forall k \in \mathsf{region}(i).\kappa \sqsubseteq \mathcal{A}_k \end{array}$$

We prove $\beta \vdash s_1' \sim^\lambda s_2'$. The proof is similar to the previous case (the only change is i' by $i+1$).

3. By the operational semantics we know:

$$\frac{B(i) = \texttt{if } i' \quad v_1 = 0}{\langle i, \alpha_1, v_1 \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i', \alpha_1, \sigma_1, \eta_1 \rangle = s_1'}$$

$$\frac{B(i) = \texttt{if } i' \quad v_2 \neq 0}{\langle i, \alpha_2, v_2 \cdot \sigma_2, \eta_2 \rangle \longrightarrow_B \langle i+1, \alpha_2, \sigma_2, \eta_2 \rangle = s_2'}$$

From T-If:

$$\begin{array}{l} \mathcal{V}_i \leq \mathcal{V}_{i'}, \mathcal{V}_{i+1} \\ \mathcal{S}_i \leq_{\mathcal{A}_i} \kappa \cdot \mathcal{S}_{i'}, \kappa \cdot \mathcal{S}_{i+1} \\ \mathcal{A}_i \sqsubseteq \kappa \\ \mathcal{T}_i \leq \mathcal{T}_{i'}, \mathcal{T}_{i+1} \\ \forall k \in \mathsf{region}(i).\kappa \sqsubseteq \mathcal{A}_k \end{array}$$

Now we prove $\beta \vdash s_1' \sim^\lambda s_2'$.

a) Since $v_1 = 0$, $v_2 \neq 0$ and $\beta, \kappa \cdot \mathcal{S}_i, \mathcal{A}_i \vdash v_1 \cdot \sigma_1 \sim^\lambda v_2 \cdot \sigma_2$, it must be the case that $\kappa \not\sqsubseteq \lambda$).
By item i) of SOAP property and condition 4 of T-If, $\kappa \sqsubseteq \mathcal{A}_{i'}, \mathcal{A}_{i+1}$. Therefore, $\mathcal{A}_{i'}, \mathcal{A}_{i+1} \not\sqsubseteq \lambda$.
b) We must prove $\beta, (\mathcal{V}_{i'}, \mathcal{V}_{i+1}), l \vdash \alpha_1 \sim^\lambda \alpha_2$, $l \not\sqsubseteq \lambda$. For any $x \in \mathbb{X}$, if either $\mathcal{V}_{i'}(x) \not\sqsubseteq \mathtt{L}$ or $\mathcal{V}_{i+1}(x) \not\sqsubseteq \mathtt{L}$, then we are done. Otherwise, $\beta, \lfloor \mathcal{V}_{i'}(x) \sqcup \mathcal{V}_{i+1}(x) \rfloor \vdash \alpha_1(x) \sim^\lambda \alpha_2(x)$ holds by the hypothesis $\beta, \mathcal{V}_i, \mathcal{A}_i \vdash \alpha_1 \sim^\lambda \alpha_2$ condition 1 of T-If.
c) By hypothesis $\beta, \mathcal{S}_i, \mathcal{A}_i \vdash v_1 \cdot \sigma_1 \sim^\lambda v_2 \cdot \sigma_2$. From this, by H-Low, $\beta, (\mathcal{S}_i, \mathcal{S}_i), l \vdash v_1 \cdot \sigma_1 \sim^\lambda v_2 \cdot \sigma_2$, $l \not\sqsubseteq \lambda$.
Now, by Lemma A.7 and Lemma A.8 we have $\beta, (\kappa \cdot \mathcal{S}_{i'}, \kappa \cdot \mathcal{S}_{i+1}), l \vdash v_1 \cdot \sigma_1 \sim^\lambda v_2 \cdot \sigma_2$.
The result, $\beta, (\mathcal{S}_{i'}, \mathcal{S}_{i+1}), l \vdash \sigma_1 \sim^\lambda \sigma_2$, follow of the definition of stack indistinguishability.
d) Idem to T-Goto case.

4. The proof the last case is similar to the previous case.

Case: Suppose $B(i) = \texttt{new}\ C$. Then

$$\frac{B(i) = \texttt{new}\ C \quad o_1 = \mathsf{Fresh}(\eta_1)}{\langle i, \alpha_1, \sigma_1, \eta_1\rangle \longrightarrow_B \langle i+1, \alpha_1, o_1 \cdot \sigma_1, \eta_1 \oplus \{o_1 \mapsto \mathsf{Default}(C)\}\rangle = s'_1}$$

$$\frac{B(i) = \texttt{new}\ C \quad o_2 = \mathsf{Fresh}(\eta_2)}{\langle i, \alpha_2, \sigma_2, \eta_2\rangle \longrightarrow_B \langle i+1, \alpha_2, o_2 \cdot \sigma_2, \eta_2 \oplus \{o_2 \mapsto \mathsf{Default}(C)\}\rangle = s'_2}$$

Moreover, from T-New:

$$\begin{array}{l} a = \mathsf{Fresh}(\mathcal{T}_i) \\ \mathcal{T}_i \oplus \{a \mapsto [f_1 : \mathtt{ft}_{\mathcal{A}_i}(f_1), \ldots, f_n : \mathtt{ft}_{\mathcal{A}_i}(f_n)]\} \leq \mathcal{T}_{i+1} \\ \langle \{a\}, \mathcal{A}_i\rangle \cdot \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \end{array}$$

We prove $\beta' \vdash s'_1 \sim^\lambda s'_2$ for $\beta' = \beta \oplus \{o_1 \mapsto o_2\}$ and $\beta'^{\lhd} = \beta^{\lhd} \cup \{a \mapsto o_1\}$ and $\beta'^{\rhd} = \beta^{\rhd} \cup \{a \mapsto o_2\}$. We prove the four points of state indistinguishability:

1. From $i'_1 = i+1 = i'_2$ it follows that $\mathcal{A}_{i'_1} = \mathcal{A}_{i'_1}$.
2. Idem to T-Goto case.
3. By hypothesis $\beta, \mathcal{S}_i, \mathcal{A}_i \vdash \sigma_1 \sim^\lambda \sigma_2$ and given that $\beta \subseteq \beta'$, we resort to lemma A.6 to deduce
$$\beta', \mathcal{S}_i, \mathcal{A}_i \vdash \sigma_1 \sim^\lambda \sigma_2 \tag{A.20}$$
We have two cases:
   - $\mathcal{A}_{i+1} \sqsubseteq \lambda$. Since $\beta', \lfloor\langle\{a\}, \mathcal{A}_i\rangle\rfloor \vdash o_1 \sim^\lambda o_2$ we appeal to L-Cons and (A.20) to deduce $\beta', \langle\{a\}, \mathtt{L}\rangle \cdot \mathcal{S}_i, \mathcal{A}_i \vdash o_1 \cdot \sigma_1 \sim^\lambda o_2 \cdot \sigma_2$. This result and condition 3 of T-New and lemma A.6 (again) allows to conclude $\beta', \mathcal{S}_{i+1}, \mathcal{A}_{i+1} \vdash o_1 \cdot \sigma_1 \sim^\lambda o_2 \cdot \sigma_2$.
   - $\mathcal{A}_{i+1} \not\sqsubseteq \lambda$. From the previous case, resorting to H-Low.
4. We must prove the two items of the definition of heap indistinguishability. The first clearly follow from the definition of β':
   a) $\mathsf{Dom}(\beta') \subseteq \mathsf{Dom}(\eta'_1)$ and $\mathsf{Ran}(\beta') \subseteq \mathsf{Dom}(\eta'_2)$.
   b) $\mathsf{Ran}(\beta'^{\lhd}) \subseteq \mathsf{Dom}(\eta'_1)$ and $\mathsf{Ran}(\beta'^{\rhd}) \subseteq \mathsf{Dom}(\eta'_2)$.
   c) $\mathsf{Dom}(\beta'^{\lhd}) \subseteq \mathsf{Dom}(\mathcal{T}_{i+1})$ and $\mathsf{Dom}(\beta'^{\rhd}) \subseteq \mathsf{Dom}(\mathcal{T}_{i+1})$.
   d) $\mathsf{Dom}(\eta'_1(o)) = \mathsf{Dom}(\eta'_2(\beta'(o)))$

   We focus on the second item, namely
$$\beta', \lfloor \mathcal{T}_{i+1}(\beta'^{\lhd}(o), f)\rfloor \vdash \eta'_1(o,f) \sim^\lambda \eta'_2(\beta'(o), f)$$
   Let $o \in \mathsf{Dom}(\beta')$ and $f \in \mathsf{Dom}(\eta'_1(o))$. By hypothesis we know $\beta, \lfloor \mathcal{T}_i(\beta^{\lhd}(o), f)\rfloor \vdash \eta_1(o,f) \sim^\lambda \eta_2(\beta(o), f)$. Also, by condition 2 of T-New and lemma A.1, $\beta', \lfloor \mathcal{T}_{i+1}(\beta^{\lhd}(o), f)\rfloor \vdash \eta_1(o,f) \sim^\lambda \eta_2(\beta(o), f)$.
   We conclude by noticing that we (trivially) have $\beta', \lfloor \mathcal{T}_{i+1}(\beta'^{\lhd}(o_1), f)\rfloor \vdash \{o_1 \mapsto \mathsf{Default}(C)\}(o_1, f) \sim^\lambda \{o_2 \mapsto \mathsf{Default}(C)\}(\{o_1 \mapsto o_2\}(o_1), f)$.

Cases: $B(i) = \texttt{push}\ v$ and $B(i) = \texttt{prim}$ are similar to case $B(i) = \texttt{load}\ x$.

Case: $B(i) = \texttt{pop}$ is similar to case $B(i) = \texttt{store}\ x$.

Case: Suppose that $B(i) = \texttt{putfield}\ f$.
There are several options to consider depending on whether exceptions are raised or not and whether there is a handler or not.

1. If no exceptions are raised, then

$$\frac{B(i) = \texttt{putfield}\ f \quad o_1 \in \mathsf{Dom}(\eta_1) \quad f \in \mathsf{Dom}(\eta_1(o_1)) \quad \mathsf{Handler}(i)\downarrow}{\langle i, \alpha_1, v_1 \cdot o_1 \cdot \sigma_1, \eta_1\rangle \longrightarrow_B \langle i+1, \alpha_1, \sigma_1, \eta_1 \oplus (\{o_1 \mapsto \eta_1(o)\} \oplus \{f_j \mapsto v_1\})\rangle = s_1'}$$

$$\frac{B(i) = \texttt{putfield}\ f \quad o_2 \in \mathsf{Dom}(\eta_2) \quad f \in \mathsf{Dom}(\eta_2(o_2)) \quad \mathsf{Handler}(i)\downarrow}{\langle i, \alpha_2, v_2 \cdot o_2 \cdot \sigma_2, \eta_2\rangle \longrightarrow_B \langle i+1, \alpha_2, \sigma_2, \eta_2 \oplus (\{o_2 \mapsto \eta_2(o)\} \oplus \{f_j \mapsto v_2\})\rangle = s_2'}$$

Moreover, from T-Ptfld:

$$\begin{array}{l}
\mathcal{S}_i \leq_{\mathcal{A}_i} \langle R_1, l_1\rangle \cdot \langle R, l\rangle \cdot \mathcal{S}_{i+1}\\
l_1 \sqcup l \sqsubseteq \prod_{a\in R} \mathcal{T}_i(a, f)\\
for\ all\ a \in R.\mathcal{T}_{i+1}(a, f) = \langle R_3, l_2\rangle,\\
\qquad\qquad where\ \mathcal{T}_i(a, f) = \langle \_, l_2\rangle \wedge R_1 \subseteq R_3\\
\mathcal{A}_i \sqsubseteq l, l_1\\
\mathcal{V}_i \leq \mathcal{V}_{i+1}\\
\mathcal{T}_i\backslash\{(a, f) \mid a \in R\} \leq \mathcal{T}_{i+1}\backslash\{(a, f) \mid a \in R\}
\end{array}$$

By setting $\beta' = \beta$ we may prove $\beta \vdash s_1' \sim^\lambda s_2'$, as developed below.

a) From $i_1' = i + 1 = i_2'$ it follows that $\mathcal{A}_{i_1'} = \mathcal{A}_{i_1'}$.
b) Idem to T-Goto case.
c) We have two cases:
   - $\mathcal{A}_{i+1} \sqsubseteq \lambda$. By hypothesis $\beta, \mathcal{S}_i, \mathcal{A}_i \vdash v_1 \cdot o_1 \cdot \sigma_1 \sim^\lambda v_2 \cdot o_2 \cdot \sigma_2$ and by condition 1 of T-Ptfld and lemma A.6, $\beta, \langle R_1, l_1\rangle \cdot \langle R, l\rangle \cdot S_{i+1}, \mathcal{A}_i \vdash v_1 \cdot o_1 \cdot \sigma_1 \sim^\lambda v_2 \cdot o_2 \cdot \sigma_2$. We conclude by resorting to the definition of stack indistinguishability and Lemma A.9: $\beta, S_{i+1}, \mathcal{A}_{i+1} \vdash \sigma_1 \sim^\lambda \sigma_2$.
   - $\mathcal{A}_{i+1} \not\sqsubseteq \lambda$. By the previous case and resorting to H-Low.
d) We prove heap indistinguishability to deduce $\beta, \mathcal{T}_{i+1} \vdash \eta_1' \sim^\lambda \eta_2'$.
   The first item is an immediate consequence of the hypothesis. For the last item we proceed as follows.
   Let $o \in \mathsf{Dom}(\beta_{loc})$, and let $f \in \mathsf{Dom}(\eta_1'(o))$. We must prove that:
   $$\beta, \lfloor \mathcal{T}_{i+1}(\beta^{\lhd-1}(o), f)\rfloor \vdash \eta_1'(o, f) \sim^\lambda \eta_2'(\beta(o), f).$$
   Recall that by hypothesis we know $\beta, \lfloor \mathcal{T}_i(\beta^{\lhd-1}(o), f)\rfloor \vdash \eta_1(o, f) \sim^\lambda \eta_2(\beta(o), f)$.
   We have two cases:
   i. if o is such that $\beta^{\lhd-1}(o) \in \mathsf{Dom}(\mathcal{T}_i \setminus R)$, the result follows from $\mathcal{T}_i \leq \mathcal{T}_{i+1}$ and lemma A.1.
   ii. if o is such that $\beta^{\lhd-1}(o) \in \mathsf{Dom}(R)$, the result follows from $\beta, l_1 \vdash v_1 \sim^\lambda v_2$, condition 2 of T-Ptfld and lemma A.1.

2. If both runs raise an exception and there is a handler, then

$$\frac{\begin{array}{c}B(i) = \texttt{putfield}\ f \quad o_1 = null\\ o_1' = \mathsf{Fresh}(\eta_1) \quad \mathsf{Handler}(i) = i'\end{array}}{\langle i', \alpha_1, v_1 \cdot o_1 \cdot \sigma_1, \eta_1\rangle \longrightarrow_B \langle i', \alpha_1, o_1' \cdot \epsilon, \eta_1 \oplus \{o_1' \mapsto \mathsf{Default}(\mathbf{Throwable})\}\rangle = s_1'}$$

$$\frac{\begin{array}{c}B(i) = \texttt{putfield}\ f \quad o_2 = null\\ o_2' = \mathsf{Fresh}(\eta_2) \quad \mathsf{Handler}(i) = i'\end{array}}{\langle i', \alpha_2, v_2 \cdot o_2 \cdot \sigma_2, \eta_2\rangle \longrightarrow_B \langle i', \alpha_2, o_2' \cdot \epsilon, \eta_2 \oplus \{o_2' \mapsto \mathsf{Default}(\mathbf{Throwable})\}\rangle = s_2'}$$

Moreover, from T-PtFldH:

$$\begin{array}{l}
\mathsf{Handler}(i) = i' \\
\mathcal{S}_i = \kappa' \cdot \langle R, l\rangle \cdot S, \text{ for some } S \\
a = \mathsf{Fresh}(\mathcal{T}_i) \\
\langle \{a\}, l \sqcup \mathcal{A}_i\rangle \cdot \epsilon \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\
\mathcal{V}_i \leq \mathcal{V}_{i'} \\
\mathcal{T}_i \oplus \{a \mapsto [f_1 : \mathtt{ft}_{\mathcal{A}_i}(f_1), \ldots, f_n : \mathtt{ft}_{\mathcal{A}_i}(f_n)]\} \leq \mathcal{T}_{i'} \\
\forall j \in \mathsf{Dom}(\mathcal{S}_i). \mathcal{A}_i \sqsubseteq \mathcal{S}_i(j)
\end{array}$$

If $l \sqcup \mathcal{A}_i \sqsubseteq \lambda$, then we set $\beta' = \beta \oplus \{o'_1 \mapsto o'_2\}$ and $\beta'^{\lhd} = \beta^{\lhd} \cup \{a \mapsto o'_1\}$ and $\beta'^{\rhd} = \beta^{\rhd} \cup \{a \mapsto o'_2\}$. We can proceed similarly to case `new`.
If $l \sqcup \mathcal{A}_i \not\sqsubseteq \lambda$, then we set to $\beta' = \beta$. Also, we can proceed similarly to case `new` in the firsts three cases. We prove heap indistinguishability to deduce $\beta', \mathcal{T}_{i'} \vdash \eta_1 \oplus \{o'_1 \mapsto \mathsf{Default}(\mathbf{Throwable})\} \sim^{\lambda} \eta_2 \oplus \{o'_2 \mapsto \mathsf{Default}(\mathbf{Throwable})\}$. The first item of heap indistinguishability result by β' definition. Now, by hypothesis $\beta', \mathcal{T}_i \vdash \eta_1 \sim^{\lambda} \eta_2$. and $o'_1 \notin \mathsf{Dom}(\beta)$ (o_1 is fresh in η_1) then trivially
$\beta', \mathcal{T}_i \oplus \{a \mapsto [\ldots]\} \vdash \eta_1 \oplus \{o'_1 \mapsto \mathsf{Default}(\mathbf{Throwable})\} \sim^{\lambda} \eta_2 \oplus \{o'_2 \mapsto \mathsf{Default}(\mathbf{Throwable})\}$.
The result follows by condition 6 of T-PtFldH and Lemma A.10.

3. If both runs raise an exception and no handler is available. The proof the last case is similar to the previous case.
4. One run is normal and an exception is raised in the other execution (for which there is a handler).

$$\frac{B(i) = \mathtt{putfield}\ f \quad o_1 \in \mathsf{Dom}(\eta_1) \quad f \in \mathsf{Dom}(\eta_1(o_1))}{\langle i, \alpha_1, v_1 \cdot o_1 \cdot \sigma_1, \eta_1\rangle \longrightarrow_B \langle i+1, \alpha_1, \sigma_1, \eta_1 \oplus (\{o_1 \mapsto \eta_1(o)\} \oplus \{f_j \mapsto v_1\})\rangle = s'_1}$$

$$\frac{\begin{array}{c} B(i) = \mathtt{putfield}\ f \quad o_2 = null \\ o' = \mathsf{Fresh}(\eta) \quad \mathsf{Handler}(i) = i' \end{array}}{\langle i, \alpha, v_2 \cdot o_2 \cdot \sigma_2, \eta_2\rangle \longrightarrow_B \langle i', \alpha_2, o' \cdot \epsilon, \eta_2 \oplus \{o' \mapsto \mathsf{Default}(\mathbf{Throwable})\}\rangle = s'_2}$$

Moreover, from T-PtFld and T-PtFldH:

$$\begin{array}{l}
\mathcal{S}_i \leq_{\mathcal{A}_i} \langle R_1, l_1\rangle \cdot \langle R, l\rangle \cdot \mathcal{S}_{i+1} \\
l_1 \sqcup l \sqsubseteq \bigsqcap_{a \in R} \mathcal{T}_i(a, f) \\
\textit{for all } a \in R. \mathcal{T}_{i+1}(a, f) = \langle R_3, l_2\rangle, \\
\qquad\qquad \textit{where } \mathcal{T}_i(a, f) = \langle \_, l_2\rangle \wedge R_1 \subseteq R_3 \\
\mathcal{A}_i \sqsubseteq l, l_1 \\
\mathcal{V}_i \leq \mathcal{V}_{i+1}, \mathcal{V}_{i'} \\
\mathcal{T}_i \backslash \{(a, f) \mid a \in R\} \leq \mathcal{T}_{i+1} \backslash \{(a, f) \mid a \in R\} \\
\forall k \in \mathtt{region}(i). l \sqsubseteq \mathcal{A}_k \\
\mathsf{Handler}(i) = i' \\
\mathcal{S}_i = \kappa' \cdot \langle R, l\rangle \cdot S, \text{ for some } S \\
a = \mathsf{Fresh}(\mathcal{T}_i) \\
\langle \{a\}, l \sqcup \mathcal{A}_i\rangle \cdot \epsilon \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\
\mathcal{T}_i \oplus \{a \mapsto [f_1 : \mathtt{ft}_{\mathcal{A}_i}(f_1), \ldots, f_n : \mathtt{ft}_{\mathcal{A}_i}(f_n)]\} \leq \mathcal{T}_{i'} \\
\forall j \in \mathsf{Dom}(\mathcal{S}_i). \mathcal{A}_i \sqsubseteq \mathcal{S}_i(j)
\end{array}$$

We set $\beta' = \beta$ and prove $\beta' \vdash s'_1 \sim^{\lambda} s'_2$.

a) From hypothesis $\beta, \mathcal{S}_i, \mathcal{A}_i \vdash v_1 \cdot o_1 \cdot \sigma_1 \sim^{\lambda} v_2 \cdot o_2 \cdot \sigma_2$ and the operational semantics $o_1 \neq null$ and $o_2 = null$. Therefore $\beta, \lfloor l \rfloor \vdash o_1 \sim^{\lambda} o_2$, i.e. $l \not\sqsubseteq \lambda$. By SOAP, $i+1, i' \in \mathsf{region}(i)$ and by condition 5 of T-PtFld $\lambda \sqsubseteq \mathcal{A}_{i+1}, \mathcal{A}_{i'}$. Then $\mathcal{A}_{i+1}, \mathcal{A}_{i'} \not\sqsubseteq \lambda$.
b) Idem to case `If` with $v_1 = 0$ and $v_2 \neq 0$.

c) We must prove that $\beta', (\mathcal{S}_{i+1}, \mathcal{S}_{i'}), l' \vdash \sigma_1 \sim^\lambda o' \cdot \epsilon$, $l' \not\sqsubseteq \lambda$.
By L-Nil and H-Low, we can affirm that $\beta', (\epsilon, \epsilon), l' \vdash \epsilon \sim^\lambda \epsilon$. Now, by $l \not\sqsubseteq \lambda$ and H-Cons-R we have that $\beta', (\epsilon, \langle\{a\}, l \sqcup \mathcal{A}_i\rangle \cdot \epsilon), l' \vdash \epsilon \sim^\lambda o' \cdot \epsilon$
By first condition of T-PtFld and the last one of T-PtFldH (both of which are listed above), we can repeatedly apply H-Cons-L to obtain $\beta', (\mathcal{S}_{i+1}, \langle\{a\}, l \sqcup \mathcal{A}_i\rangle \cdot \epsilon), l' \vdash \sigma_1 \sim^\lambda o' \cdot \epsilon$. The this and condition 11 of T-PtFld we can apply Lemma A.7 and result holds.
d) We prove items of heap indistinguishability to deduce $\beta', (\mathcal{T}_{i+1}, \mathcal{T}_{i'}) \vdash \eta_1 \oplus (\{o_1 \mapsto \eta_1(o)\} \oplus \{f_j \mapsto v_1\}) \sim^\lambda \eta_2 \oplus \{o' \mapsto \mathsf{Default}(\mathbf{Throwable})\}$.
The first item is an immediate consequence of the hypothesis. The last item follows from the hypothesis and $o' \notin \mathsf{Dom}(\beta'_{loc})$ (it is a high object). Furthermore, by condition 2, the updated field is high.

5. One execution is normal and the other one raises an exception (for which there is no handler). The proof is similar to previous case.

Case: $B(i) = \texttt{throw}$
There are two options to consider:

1. If there is a handler, then

$$\frac{B(i) = \texttt{throw} \quad \mathsf{Handler}(i) = i'}{\langle i, \alpha_1, o_1 \cdot \sigma_1, \eta_1\rangle \longrightarrow_B \langle i', \alpha_1, o_1 \cdot \epsilon, \eta_1\rangle = s'_1}$$

$$\frac{B(i) = \texttt{throw} \quad \mathsf{Handler}(i) = i'}{\langle i, \alpha_2, o_2 \cdot \sigma_2, \eta_2\rangle \longrightarrow_B \langle i', \alpha_2, o_2 \cdot \epsilon, \eta_2\rangle = s'_2}$$

Moreover, by T-ThrH,

$$\begin{array}{l} \mathsf{Handler}(i) = i' \\ \mathcal{S}_i(0) \sqsubseteq \mathcal{A}_{i'} \\ \mathcal{S}_i(0) \cdot \epsilon \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\ \mathcal{V}_i \leq \mathcal{V}_{i'} \\ \mathcal{T}_i \leq \mathcal{T}_{i'} \\ \forall j \in \mathsf{Dom}(\mathcal{S}_i).\mathcal{A}_i \sqsubseteq \mathcal{S}_i(j) \end{array}$$

Set $\beta' = \beta$. We prove $\beta \vdash s'_1 \sim^\lambda s'_2$ below.
a) By the operational semantics $i'_1 = i' = i'_2$ and hence $\mathcal{A}_{i'_1} = \mathcal{A}_{i'_1}$.
b) Idem to T-Goto case.
c) We have two cases:
- $\mathcal{A}_{i'} \sqsubseteq \lambda$. By hypothesis $\beta, \mathcal{S}_i, \mathcal{A}_i \vdash o_1 \cdot \sigma_1 \sim^\lambda o_2 \cdot \sigma_2$ then $\beta, \lfloor\mathcal{S}_i(0)\rfloor \vdash o_1 \sim^\lambda o_2$. By this and by L-Nil we can apply L-Cons for obtain the result.
- $\mathcal{A}_{i'} \not\sqsubseteq \lambda$. By the previous case and resorting to H-Low.

d) Idem to T-Goto case.

2. If there is no handler, then proof is idem to T-Goto case.

Cases: $B(i) = \texttt{getfield}\ f$.
This case is is similar to case $B(i) = \texttt{putfield}$.

Case: We assume $B(i) = \texttt{return}$. By the operational semantics we know:

$$\frac{B(i_1) = \texttt{return}}{\langle i, \alpha_1, v_1 \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle p_f, v_1, \eta_1 \rangle = s'_1}$$

$$\frac{B(i_2) = \texttt{return}}{\langle i, \alpha_2, v_2 \cdot \sigma_2, \eta_2 \rangle \longrightarrow_B \langle p_f, v_2, \eta_2 \rangle = s'_2}$$

From T-Ret:

$$\begin{array}{l} \mathcal{S}_i(0) \sqcup \mathcal{A}_i \sqsubseteq \kappa_r \\ \mathcal{T}_i \leq \mathcal{T}_{p_f} \end{array}$$

We take $\beta' = \beta$ and prove $\beta \vdash s'_1 \sim^\lambda s'_2$.

1. We must prove $\beta, \kappa_r \vdash v_1 \sim^\lambda v_2$. By the hypothesis, $\beta, \mathcal{S}_i, \mathcal{A}_i \vdash v_1 \cdot \sigma_1 \sim^\lambda v_2 \cdot \sigma_2$ then we have $\beta, \lfloor \mathcal{S}_i(0) \rfloor \vdash v_1 \sim^\lambda v_2$. Furthermore, by T-Ret $\mathcal{S}_i(0) \sqsubseteq \kappa_r$. Now, by indistinguishability definition $\beta, \kappa_r \vdash v_1 \sim^\lambda v_2$.
2. $\eta_1 \sim^\lambda_{\beta, \mathcal{T}_{p_f}, \mathcal{A}_{p_f}} \eta_2$ follow by $\mathcal{T}_{i_1} \leq \mathcal{T}_{p_f}$ and Lemma A.10.

Cases: We assume $\texttt{pc}(i) = err_i$. By the operational semantics we know:

$$\frac{B(err_i)}{\langle err_i, \alpha_1, \langle o_1 \rangle \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle p_f, \langle o_1 \rangle, \eta_1 \rangle = s'_1}$$

$$\frac{B(err_i)}{\langle err_i, \alpha_2, \langle o_2 \rangle \cdot \sigma_2, \eta_2 \rangle \longrightarrow_B \langle p_f, \langle o_2 \rangle, \eta_2 \rangle = s'_2}$$

From T-Ret:

$$\begin{array}{l} \mathcal{S}_{err_i}(0) \sqcup \mathcal{A}_{err_i} \sqsubseteq \kappa_r \\ \mathcal{T}_{err_i} \leq \mathcal{T}_{p_f} \end{array}$$

The proof is idem to T-Ret case.

Cases: We assume $\texttt{pc}(i) = \texttt{invokevirtual}\ m$. There are several options to consider depending on whether exceptions are raised or not and whether there is a handler or not.

1. If no exceptions are raised, then

$$\frac{\begin{array}{c} B(i) = \texttt{invokevirtual}\ m \quad o_1 \in \mathsf{Dom}(\eta_1) \quad B' = \mathsf{lookup}(m, \mathsf{dynamic}(o_1)) \\ \langle 1, \{this \mapsto o_1, \overrightarrow{x \mapsto v_1}\}, \epsilon, \eta_1 \rangle \longrightarrow^*_{B'} \langle p'_f, v_1, \eta'_1 \rangle \end{array}}{\langle i, \alpha_1, \overrightarrow{v_1} \cdot o_1 \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i+1, \alpha_1, v_1 \cdot \sigma_1, \eta'_1 \rangle = s'_1}$$

$$\frac{\begin{array}{c} B(i) = \texttt{invokevirtual}\ m \quad o_2 \in \mathsf{Dom}(\eta_2) \quad B' = \mathsf{lookup}(m, \mathsf{dynamic}(o_2)) \\ \langle 1, \{this \mapsto o_2, \overrightarrow{x \mapsto v_2}\}, \epsilon, \eta_2 \rangle \longrightarrow^*_{B'} \langle p'_f, v_2, \eta'_2 \rangle \end{array}}{\langle i, \alpha_2, \overrightarrow{v_2} \cdot o_2 \cdot \sigma_2, \eta_2 \rangle \longrightarrow_B \langle i+1, \alpha_2, v_2 \cdot \sigma_2, \eta'_2 \rangle = s'_2}$$

Moreover, from T-InvkVrt:

$$\begin{array}{l}
\mathsf{Handler}_B(i)\downarrow \\
\mathsf{mt}_m(\kappa \sqcup \mathcal{A}_i) = \mathcal{A}'_1, \mathcal{T}'_1, \mathcal{T}'_{p_f} \rhd ((\overrightarrow{x' : \kappa'}, \kappa'_r, E'), B') \\
\mathcal{A}_i \sqcup \kappa = \mathcal{A}'(1) \\
\mathcal{S}_i = \kappa'_1 \cdots \kappa'_n \cdot \kappa \cdot S, \text{ for some } S \\
\mathcal{S}_i \leq_{\mathcal{A}_i} \kappa'_1 \cdots \kappa'_n \cdot \kappa \cdot (\mathcal{S}_{i+1} \backslash 0) \\
\mathcal{A}_i \sqsubseteq \kappa'_1, \cdots, \kappa'_n, \kappa \\
\kappa \sqcup \kappa'_r \sqcup \mathcal{A}_i \sqsubseteq \mathcal{S}_{i+1}(0) \\
\mathcal{V}_i \leq \mathcal{V}_{i+1} \\
\mathcal{T}_i \leq \mathcal{T}'_1 \leq \mathcal{T}'_{p_f} \leq \mathcal{T}_{i+1} \\
\forall k \in \mathtt{region}(i). \kappa \sqcup \kappa'_r \sqsubseteq \mathcal{A}_k, \text{ if } i \in \mathsf{Dom}(B^\sharp)
\end{array}$$

By setting $\beta' \supseteq \beta$, where β' is the location bijection set such that $\lambda \vdash \langle p'_f, v_1, \eta'_1 \rangle \sim^{\beta'} \langle p'_f, v_2, \eta'_2 \rangle$, we may prove $\lambda \vdash s'_1 \sim^{\beta'} s'_2$, as developed below. The proof that exist β' proceeds by induction on the quantity of INVOKEVITUAL instructions in the derivation for $s \longrightarrow_B s'$.

a) From $i'_1 = i + 1 = i'_2$ it follows that $\mathcal{A}(i'_1) = \mathcal{A}(i'_2)$.

b) By hypothesis $\beta, \mathcal{V}_i, \mathcal{A}_i \vdash \alpha_1 \sim^\lambda \alpha_2$, condition 8 of T-InvkVrt and lemma A.2, $\beta', \mathcal{V}_{i+1}, \mathcal{A}_i \vdash \alpha_1 \sim^\lambda \alpha_2$. If $\mathcal{A}(i+1) \sqsubseteq \lambda$, the result follows by Lemma A.5.
In the case that $\mathcal{A}(i+1) \not\sqsubseteq \lambda$ we appeal to the definition of indistinguishability of local variables to deduce $\beta', \mathcal{V}_{i+1}, \mathcal{A}_{i+1} \vdash \alpha_1 \sim^\lambda \alpha_2$.

c) We have two cases:
- $\mathcal{A}(i+1) \sqsubseteq \lambda$. By hypothesis $\beta, \mathcal{S}_i, \mathcal{A}_i \vdash \overrightarrow{v_1} \cdot o_1 \cdot \sigma_1 \sim^\lambda \overrightarrow{v_2} \cdot o_2 \cdot \sigma_2$ and by condition 5 of T-InvkVrt and lemma A.6, $\beta', \kappa'_1 \cdots \kappa'_n \cdot \kappa \cdot (\mathcal{S}_{i+1} \backslash 0), \mathcal{A}_i \vdash \overrightarrow{v_1} \cdot o_1 \cdot \sigma_1 \sim^\lambda \overrightarrow{v_2} \cdot o_2 \cdot \sigma_2$. By resorting to the definition of stack indistinguishability we have $\beta', \mathcal{S}_{i+1} \backslash 0, \mathcal{A}_i \vdash \sigma_1 \sim^\lambda \sigma_2$. Now, by hypothesis $\beta' \vdash \langle p_f, v_1, \eta'_1 \rangle \sim^\lambda \langle p_f, v_2, \eta'_2 \rangle$ and condition 7 of T-InvkVrt and L-Cons. we conclude $\beta', \mathcal{S}_{i+1}, \mathcal{A}_i \vdash v_1 \cdot \sigma_1 \sim^\lambda v_2 \cdot \sigma_2$. The result follows by Lemma A.9.
- $\mathcal{A}(i+1) \not\sqsubseteq \lambda$. By the previous case and resorting to H-Low.

d) By hypothesis $\beta', \mathcal{T}'_{p_f} \vdash \eta'_1 \sim^\lambda \eta'_2$, condition 9 of T-InvkVrt and Lemma A.10, $\beta', \mathcal{T}_{i+1} \vdash \eta'_1 \sim^\lambda \eta'_2$.

2. If both runs raise an exception and there is a handler, then

$$\frac{\begin{array}{c} B(i) = \mathtt{invokevirtual}\ m \quad o'_1 = \mathsf{Fresh}(\eta_1) \quad B' = \mathsf{lookup}(m, \mathsf{dynamic}(o_1)) \\ \langle 1, \{this \mapsto o_1, \overrightarrow{x \mapsto v_1}\}, \epsilon, \eta_1 \rangle \longrightarrow^*_{B'} \langle p'_f, \langle o'_1 \rangle, \eta'_1 \rangle \quad \mathsf{Handler}_B(i) = i' \end{array}}{\langle i, \alpha_1, \overrightarrow{v_1} \cdot o_1 \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i', \alpha_1, \langle o'_1 \rangle \cdot \epsilon, \eta'_1 \rangle = s'_1}$$

$$\frac{\begin{array}{c} B(i) = \mathtt{invokevirtual}\ m \quad o'_2 = \mathsf{Fresh}(\eta_2) \quad B' = \mathsf{lookup}(m, \mathsf{dynamic}(o_2)) \\ \langle 1, \{this \mapsto o_2, \overrightarrow{x \mapsto v_2}\}, \epsilon, \eta_2 \rangle \longrightarrow^*_{B'} \langle p'_f, \langle o'_2 \rangle, \eta'_2 \rangle \quad \mathsf{Handler}_B(i) = i' \end{array}}{\langle i, \alpha_2, \overrightarrow{v_2} \cdot o_2 \cdot \sigma_2, \eta_2 \rangle \longrightarrow_B \langle i', \alpha_2, \langle o'_2 \rangle \cdot \epsilon, \eta'_2 \rangle = s'_2}$$

Moreover, from T-InvkVrtH:

$$\begin{array}{l}
\mathsf{Handler}_B(i) = i' \\
\mathsf{mt}_m(\kappa \sqcup \mathcal{A}_i) = \mathcal{A}'_1, \mathcal{T}'_1, \mathcal{T}'_{p_f} \rhd ((\overrightarrow{x' : \kappa'}, \kappa'_r, E'), B') \\
E' \neq \emptyset \\
\mathcal{A}_i \sqcup \kappa = \mathcal{A}'(1) \\
\kappa'_r \sqcup \mathcal{A}_i \cdot \epsilon \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\
\mathcal{T}_i \leq \mathcal{T}'_1 \leq \mathcal{T}'_{p_f} \leq \mathcal{T}_{i'} \\
\mathcal{V}_i \leq \mathcal{V}_{i'} \\
\forall j \in \mathsf{Dom}(\mathcal{S}_i). \mathcal{A}_i \sqsubseteq \mathcal{S}_i(j)
\end{array}$$

By setting $\beta' \supseteq \beta$, where β' is the location bijection set such that $\beta' \vdash \langle p'_f, o'_1, \eta'_1 \rangle \sim^\lambda \langle p'_f, o'_2, \eta'_2 \rangle$, we may prove $\beta' \vdash s'_1 \sim^\lambda s'_2$, as developed below.

a) From $i'_1 = i' = i'_2$ it follows that $\mathcal{A}(i'_1) = \mathcal{A}(i'_2)$.
b) Idem to previous case.
c) We have two cases:
- $\mathcal{A}(i') \sqsubseteq \lambda$. By hypothesis $\beta' \vdash \langle p'_f, o'_1, \eta'_1\rangle \sim^\lambda \langle p'_f, o'_2, \eta'_2\rangle$, L-Nil and L-Cons we have $\beta', \kappa'_r \sqcup \mathcal{A}_i \cdot \epsilon, \mathcal{A}_i \vdash o'_1 \cdot \epsilon \sim^\lambda o'_2 \cdot \epsilon$.
  Now, by condition 5 of T-InvkVrt and Lemma A.6 we have $\beta', \mathcal{S}_{i'}, \mathcal{A}_i \vdash o'_1 \cdot \epsilon \sim^\lambda o'_2 \cdot \epsilon$. The result follows by Lemma A.9.
- $\mathcal{A}(i') \not\sqsubseteq \lambda$. By the previous case and resorting to H-Low.

d) Idem to previous case.

3. If both runs raise an exception and no handler is available, then

$$\frac{B(i) = \texttt{invokevirtual}\ m \quad o'_1 = \mathsf{Fresh}(\eta_1) \quad \mathsf{Handler}_B(i)\uparrow \quad B' = \mathsf{lookup}(m, \mathsf{dynamic}(o_1)) \quad \langle 1, \{this \mapsto o_1, \overrightarrow{x \mapsto v_1}\}, \epsilon, \eta_1\rangle \longrightarrow^*_{B'} \langle p'_f, \langle o'_1\rangle, \eta'_1\rangle}{\langle i, \alpha_1, \overrightarrow{v_1} \cdot o_1 \cdot \sigma_1, \eta_1\rangle \longrightarrow_B \langle err_i, \alpha_1, \langle o'_1\rangle \cdot \sigma_1, \eta'_1\rangle = s'_1}$$

$$\frac{B(i) = \texttt{invokevirtual}\ m \quad o'_2 = \mathsf{Fresh}(\eta_2) \quad \mathsf{Handler}_B(i)\uparrow \quad B' = \mathsf{lookup}(m, \mathsf{dynamic}(o_2)) \quad \langle 1, \{this \mapsto o_2, \overrightarrow{x \mapsto v_2}\}, \epsilon, \eta_2\rangle \longrightarrow^*_{B'} \langle p'_f, \langle o'_2\rangle, \eta'_2\rangle}{\langle i, \alpha_2, \overrightarrow{v_2} \cdot o_2 \cdot \sigma_2, \eta_2\rangle \longrightarrow_B \langle err_i, \alpha_2, \langle o'_2\rangle \cdot \sigma_2, \eta'_2\rangle = s'_2}$$

Moreover, from T-InvkVrtNoH:

$$\begin{array}{l}
\mathsf{Handler}_B(i)\uparrow \\
\mathsf{mt}_m(\kappa \sqcup \mathcal{A}_i) = \mathcal{A}'_1, \mathcal{T}'_1, \mathcal{T}'_{p_f} \rhd ((\overrightarrow{x' : \kappa'}, \kappa'_r, E'), B') \\
E' \neq \emptyset \\
\mathcal{A}_i \sqcup \kappa = \mathcal{A}'(1) \\
\mathcal{S}_i \leq_{\mathcal{A}_i} \kappa'_1 \cdots \kappa'_n \cdot \kappa \cdot (\mathcal{S}_{err_i} \backslash 0) \\
\kappa'_r \sqcup \mathcal{A}_i \sqsubseteq \mathcal{S}_{err_i}(0) \\
\mathcal{A}_i \sqcup \kappa \sqcup \kappa'_r \sqsubseteq \kappa_r \\
\mathcal{V}_i \leq \mathcal{V}_{err_i} \\
\mathcal{T}_i \leq \mathcal{T}'_1 \leq \mathcal{T}'_{p_f} \leq \mathcal{T}_{err_i}
\end{array}$$

By setting $\beta' \supseteq \beta$, where β' is the location bijection set such that $\beta' \vdash \langle p'_f, o'_1, \eta'_1\rangle \sim^\lambda \langle p'_f, o'_2, \eta'_2\rangle$, we may prove $\beta' \vdash s'_1 \sim^\lambda s'_2$, as developed below.
a) From $i'_1 = err_i = i'_2$ it follows that $\mathcal{A}(i'_1) = \mathcal{A}(i'_2)$.
b) Idem to previous case.
c) Idem to both runs raise an exception and no handler is available case of `putfield` instruction.
d) Idem to previous case.

4. One run is normal and an exception is raised in the other execution (for which there is a handler).

$$\frac{B(i) = \texttt{invokevirtual}\ m \quad o_1 \in \mathsf{Dom}(\eta_1) \quad B' = \mathsf{lookup}(m, \mathsf{dynamic}(o_1)) \quad \langle 1, \{this \mapsto o_1, \overrightarrow{x \mapsto v_1}\}, \epsilon, \eta_1\rangle \longrightarrow^*_{B'} \langle p'_f, v_1, \eta'_1\rangle}{\langle i, \alpha_1, \overrightarrow{v_1} \cdot o_1 \cdot \sigma_1, \eta_1\rangle \longrightarrow_B \langle i+1, \alpha_1, v_1 \cdot \sigma_1, \eta'_1\rangle = s'_1}$$

$$\frac{B(i) = \texttt{invokevirtual}\ m \quad o' = \mathsf{Fresh}(\eta_2) \quad B' = \mathsf{lookup}(m, \mathsf{dynamic}(o_2)) \quad \langle 1, \{this \mapsto o_2, \overrightarrow{x \mapsto v_2}\}, \epsilon, \eta_2\rangle \longrightarrow^*_{B'} \langle p'_f, \langle o'\rangle, \eta'_2\rangle \quad \mathsf{Handler}_B(i) = i'}{\langle i, \alpha_2, \overrightarrow{v_2} \cdot o_2 \cdot \sigma_2, \eta_2\rangle \longrightarrow_B \langle i', \alpha_2, \langle o'\rangle \cdot \epsilon, \eta'_2\rangle = s'_2}$$

Moreover, from T-InvkVrtl and T-InvkVrtlH:

$$
\begin{array}{l}
\mathsf{Handler}_B(i) = i' \\
\mathsf{mt}_m(\kappa \sqcup \mathcal{A}_i) = \mathcal{A}'_1, \mathcal{T}'_1, \mathcal{T}'_{p_f} \rhd ((\overrightarrow{x' : \kappa'}, \kappa'_r, E'), B') \\
\mathcal{A}_i \sqcup \kappa = \mathcal{A}'(1) \\
\mathcal{S}_i = \kappa'_1 \cdots \kappa'_n \cdot \kappa \cdot S, \text{ for some } S \\
\mathcal{S}_i \leq_{\mathcal{A}_i} \kappa'_1 \cdots \kappa'_n \cdot \kappa \cdot (\mathcal{S}_{i+1} \backslash 0) \\
\mathcal{A}_i \sqsubseteq \kappa'_1, \cdots, \kappa'_n, \kappa \\
\kappa \sqcup \kappa'_r \sqcup \mathcal{A}_i \sqsubseteq \mathcal{S}_{i+1}(0) \\
\mathcal{V}_i \leq \mathcal{V}_{i+1}, \mathcal{V}_{i'} \\
\mathcal{T}_i \leq \mathcal{T}'_1 \leq \mathcal{T}'_{p_f} \leq \mathcal{T}_{i+1}, \mathcal{T}_{i'} \\
\forall k \in \mathtt{region}(i). \kappa \sqcup \kappa'_r \sqsubseteq \mathcal{A}_k, \text{ if } i \in \mathsf{Dom}(B^\sharp) \\
E' \neq \emptyset \\
\kappa'_r \sqcup \mathcal{A}_i \cdot \epsilon \leq_{\mathcal{A}_i} \mathcal{S}_{i'} \\
\forall j \in \mathsf{Dom}(\mathcal{S}_i). \mathcal{A}_i \sqsubseteq \mathcal{S}_i(j)
\end{array}
$$

By setting $\beta' \supseteq \beta$, where β' is the location bijection set such that $\beta' \vdash \langle p'_f, o'_1, \eta'_1 \rangle \sim^\lambda \langle p'_f, o'_2, \eta'_2 \rangle$, we may prove $\beta' \vdash s'_1 \sim^\lambda s'_2$, as developed below.

a) From hypothesis one run is normal and an exception is raised in the other execution (for which there is a handler) and $\beta', \lfloor \kappa'_r \rfloor \vdash v_1 \sim^\lambda o'$ with $\kappa'_r \not\sqsubseteq \lambda$. By SOAP, $i+1, i' \in \mathsf{region}(i)$ and by condition 10 of T-InvkVrtl $\mathcal{A}(i+1), \mathcal{A}(i') \not\sqsubseteq \lambda$.

b) By hypothesis $\beta, \mathcal{V}_i, \mathcal{A}_i \vdash \alpha_1 \sim^\lambda \alpha_2$, conditions 8 of T-InvkVrtl and Lemmas A.3 and A.4 we deduce $\beta', (\mathcal{V}_{i+1}, \mathcal{V}_{i'}), \mathcal{A}_1 \vdash \alpha_1 \sim^\lambda \alpha_2$. we resort to the definition of indistinguishability of local variable arrays and conditions 8 of T-InvkVrtl to deduce the result.

c) We must prove that $\beta', (\mathcal{S}_{i+1}, \mathcal{S}_{i'}), l \vdash v_1 \cdot \sigma_1 \sim^\lambda o' \cdot \epsilon$, with $l \not\sqsubseteq \lambda$.
By L-Nil and H-Low, we can affirm that $\beta', (\epsilon, \epsilon), l \vdash \epsilon \sim^\lambda \epsilon$. Now, by $\kappa'_r \not\sqsubseteq \lambda$ and H-Cons-R we have that $\beta', (\epsilon, \kappa'_r \sqcup \mathcal{A}(i) \cdot \epsilon), l \vdash \epsilon \sim^\lambda o'_2 \cdot \epsilon$
By conditions 3, 4, 5 and 6 and the last one of T-InvkVrtl (both of which are listed above), we can repeatedly apply H-Cons-L, following to application of Lemmas A.8 and A.8 to obtain $\beta', (\mathcal{S}_{i+1}, \mathcal{S}_{i'}), l \vdash v_1 \cdot \sigma_1 \sim^\lambda o'_2 \cdot \epsilon$.

d) By hypothesis $\beta', \mathcal{T}'_{p_f} \vdash \eta'_1 \sim^\lambda \eta'_2$, condition 9 of `T-InvkVrt` and Lemma A.10, $\beta', (\mathcal{T}_{i+1}, \mathcal{T}_{i'}) \vdash \eta'_1 \sim^\lambda \eta'_2$.

5. One execution is normal and the other one raises an exception (for which there is no handler). This case is similar to previous case.

∎

A.3 High Context

We address the proof of the One-Step Noninterference in High Level Environments.

Lemma 3.45 (One-Step High)
Let $i_1, i_2 \in \mathsf{region}(k)$ for some k such that $\forall k' \in \mathsf{region}(k) : \mathcal{A}_{k'} \not\sqsubseteq \lambda$. Furthermore, suppose:

1. $\beta \vdash s_1 \sim^\lambda s'_1$ for some location bijection set β;
2. $s_1 \longrightarrow_B s_2$;
3. $s_2 = \langle i'_1, \alpha'_1, \sigma'_1, \eta'_1 \rangle$; and
4. $i'_1 \in \mathsf{region}(k)$.

Then $\beta \vdash s_2 \sim^\lambda s'_1$.

Proof. First we prove $\beta^{\mathsf{id}} \vdash s_1 \sim^\lambda s'_1$ by a case analysis on the instruction that is executed and we conclude $\beta \circ \beta^{\mathsf{id}} \vdash s'_1 \sim^\lambda s_2$ by the typing rule and the third item of the Lemma 3.40 (transitivity). We note that by β^{id} definition $\beta \circ \beta^{\mathsf{id}} = \beta$ then we conclude $\beta \vdash s'_1 \sim^\lambda s_2$

Before commencing, we point out that $\mathcal{A}_{i_1}, \mathcal{A}_{i_2}, \mathcal{A}_{i'_1} \not\sqsubseteq \lambda$. In particular, $\mathcal{A}_{i_1}, \mathcal{A}_{i'_1} \not\sqsubseteq \lambda$ and thus this proves the first item needed to determined that s_1 and s'_1 are indistinguishable. We consider the remaining three for each reduction steps case.

Case: $B(i_1) = \texttt{store}\ x$

$$\frac{B(i_1) = \texttt{store}\ x}{\langle i_1, \alpha_1, v_1 \cdot \sigma_1, \eta_1\rangle \longrightarrow_B \langle i_1 + 1, \alpha_1 \oplus \{x \mapsto v_1\}, \sigma_1, \eta_1\rangle = s'_1}$$

Moreover, by T-Store:

$$\begin{array}{l} \mathcal{S}_{i_1} \leq_{\mathcal{A}_{i_1}} \mathcal{V}_{i_1+1}(x) \cdot \mathcal{S}_{i_1+1} \\ \mathcal{V}_{i_1} \setminus x \leq \mathcal{V}_{i_1+1} \setminus x \\ \mathcal{A}_{i_1} \sqsubseteq \mathcal{V}_{i_1+1}(x) \\ \mathcal{T}_{i_1} \leq \mathcal{T}_{i_1+1} \end{array}$$

We prove $\beta^{\mathsf{id}} \vdash s_1 \sim^\lambda s'_1$ by considering the two remaining items of machine state indistinguishability:

1. First we $\beta^{\mathsf{id}}, (\mathcal{V}_{i_1}, \mathcal{V}_{i_1+1}), \mathcal{A}_{i_1+1} \vdash \alpha_1 \sim^\lambda \alpha_1 \oplus \{x \mapsto v_1\}$.
   The only variable of interest is x since it is the only one that is changed. From condition 3 of T-Store, $\mathcal{A}_{i_1} \sqsubseteq \mathcal{V}_{i_1+1}(x)$ and hence $\beta^{\mathsf{id}}, \lfloor(\mathcal{V}_{i_1}(x) \sqcup \mathcal{V}_{i_1+1}(x))\rfloor \vdash \alpha_1(x) \sim^\lambda \alpha_1 \oplus \{x \mapsto v_1\}(x)$ **(1)**.
   Now, by Lemma 3.39 (reflexivity of variable high indistinguishability)
   $$\beta^{\mathsf{id}}, (\mathcal{V}_{i_1} \setminus x, \mathcal{V}_{i_1} \setminus x), \mathcal{A}_{i_1} \vdash \alpha_1 \setminus x \sim^\lambda \alpha_1 \oplus \{x \mapsto v_1\} \setminus x$$
   By this and by condition 2 of T-Store and by Lemma A.3 we have $\beta^{\mathsf{id}}, (\mathcal{V}_{i_1} \setminus x, \mathcal{V}_{i_1+1} \setminus x), \mathcal{A}_{i_1} \vdash \alpha_1 \setminus x \sim^\lambda \alpha_1 \oplus \{x \mapsto v_1\} \setminus x$ **(2)**. By **(2)**, **(1)** and indistinguishability definition, we conclude that $\beta^{\mathsf{id}}, (\mathcal{V}_{i_1}, \mathcal{V}_{i_1+1}), \mathcal{A}_{i_1+1} \vdash \alpha_1 \sim^\lambda \alpha_1 \oplus \{x \mapsto v_1\}$.
2. By Lemma A.14 (reflexivity of stack high indistinguishability) $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1} \setminus 0, \mathcal{S}_{i_1} \setminus 0), \mathcal{A}_{i_1} \vdash \sigma_1 \sim^\lambda \sigma_1$.
   By condition 1 of T-Store, $\mathcal{S}_{i_1} \setminus 0 \leq_{\mathcal{A}_{i_1}} \mathcal{S}_{i_1+1}$. Therefore, by resorting to Lemma A.7 we deduce
   $$\beta^{\mathsf{id}}, (\mathcal{S}_{i_1} \setminus 0, \mathcal{S}_{i_1+1}), \mathcal{A}_{i_1} \vdash \sigma_1 \sim^\lambda \sigma_1$$
   Finally, by H-Cons-L
   $$\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1+1}), \mathcal{A}_{i_1+1} \vdash v_1 \cdot \sigma_1 \sim^\lambda \sigma_1$$
3. By definition of β^{id} and Lemma A.15 (reflexivity of heap high indistinguishability) we have that $\beta^{\mathsf{id}}, (\mathcal{T}_{i_1}, \mathcal{T}_{i_1}) \vdash \eta_1 \sim^\lambda \eta_1$. Now, by the hypothesis $\mathcal{T}_{i_1} \leq \mathcal{T}_{i_1+1}$ and the Lemma A.11 we deduce $\beta^{\mathsf{id}}, (\mathcal{T}_{i_1}, \mathcal{T}_{i_1+1}) \vdash \eta_1 \sim^\lambda \eta_1$.

Case: $B(i_1) = \texttt{load}\ x$. By the operational semantics :

$$\frac{B(i_1) = \texttt{load}\ x}{\langle i_1, \alpha_1, \sigma_1, \eta_1\rangle \longrightarrow_B \langle i_1 + 1, \alpha_1, \alpha_1(x) \cdot \sigma_1, \eta_1\rangle = s'_1}$$

Moreover, from T-Load:

$$\begin{array}{l} \mathcal{A}_{i_1} \sqcup \mathcal{V}_{i_1}(x) \cdot \mathcal{S}_{i_1} \leq_{\mathcal{A}_{i_1}} \mathcal{S}_{i_1+1} \\ \mathcal{V}_{i_1} \leq \mathcal{V}_{i_1+1} \\ \mathcal{T}_{i_1} \leq \mathcal{T}_{i_1+1} \end{array}$$

We prove the three remaining items of $\beta^{\mathsf{id}} \vdash s_1 \sim^\lambda s'_1$.

1. We address $\beta^{\mathsf{id}}, (\mathcal{V}_{i_1}, \mathcal{V}_{i_1+1}), \mathcal{A}_{i_1+1} \vdash \alpha_1 \sim^\lambda \alpha_1$.
   By Lemma 3.39 (reflexivity of variable high indistinguishability), $\beta^{\mathsf{id}}, (\mathcal{V}_{i_1}, \mathcal{V}_{i_1}), \mathcal{A}_{i_1} \vdash \alpha_1 \sim^\lambda \alpha_1$. By this and by condition 2 of T-Load and by Lemma A.3, $\beta^{\mathsf{id}}, (\mathcal{V}_{i_1}, \mathcal{V}_{i_1+1}), \mathcal{A}_{i_1} \vdash \alpha_1 \sim^\lambda \alpha_1$.
2. By Lemma A.14 (reflexivity of stack high indistinguishability) we have that $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1}), \mathcal{A}_{i_1} \vdash \sigma_1 \sim^\lambda \sigma_1$. Given that $\mathcal{A}_{i_1} \not\sqsubseteq \lambda$ we may resort to H-Cons-R to obtain $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{A}(i) \sqcup \mathcal{V}_{i_1}(x) \cdot \mathcal{S}_{i_1})), \mathcal{A}_{i_1} \vdash \sigma_1 \sim^\lambda \alpha_1(x) \cdot \sigma_1$. Finally, by condition 1 of T-Load and Lemma A.7 we have $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1+1}), \mathcal{A}(i_1+1) \vdash \sigma_1 \sim^\lambda \alpha_1(x) \cdot \sigma_1$.
3. Idem to `store` case.

Case: $B(i) = \texttt{new}\ C$

$$\frac{B(i_1) = \texttt{new}\ C \quad o_1 = \mathsf{Fresh}(\eta_1)}{\langle i_1, \alpha_1, \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i_1+1, \alpha_1, o_1 \cdot \sigma_1, \eta_1 \oplus \{o_1 \mapsto \mathsf{Default}(C)\} \rangle = s_1'}$$

Therefore $\eta_1' = \eta_1 \oplus \{o_1 \mapsto \mathsf{Default}(C)\}$. Moreover, from T-New:

$$\begin{array}{l} a = \mathsf{Fresh}(T_{i_1}) \\ \mathcal{T}_{i_1} \oplus \{a \mapsto [f_1 : \texttt{ft}(f_1), \ldots, f_n : \texttt{ft}(f_n)]\} \leq \mathcal{T}_{i_1+1} \\ \langle \{a\}, \mathcal{A}_{i_1} \rangle \cdot \mathcal{S}_{i_1} \leq_{\mathcal{A}_{i_1}} \mathcal{S}_{i_1+1} \\ \mathcal{V}_{i_1} \leq \mathcal{V}_{i_1+1} \end{array}$$

We prove the three remaining items of $\beta^{\mathsf{id}} \vdash s_1 \sim^\lambda s_1'$.

1. Idem to `load` case.
2. By Lemma A.14 (reflexivity of stack high indistinguishability), $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1}), \mathcal{A}_{i_1} \vdash \sigma_1 \sim^\lambda \sigma_1$. By this and H-Cons-R, $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \langle \{a\}, \mathcal{A}(i_1) \rangle \cdot \mathcal{S}_{i_1}), \mathcal{A}_{i_1} \vdash \sigma_1 \sim^\lambda o_1 \cdot \sigma_1$.
   Now, by condition 3 of T-New and Lemma A.7 we conclude that $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1+1}), \mathcal{A}_{i_1+1} \vdash \sigma_1 \sim^\lambda o_1 \cdot \sigma_1$.
3. We prove the two items of $\beta^{\mathsf{id}}, (\mathcal{T}_{i_1}, \mathcal{T}_{i_1+1}) \vdash \eta_1 \sim^\lambda \eta_1'$.
   The first is an immediate consequence of the definition of β^{id}.
   The last item is proved as follows. Let $o \in \mathsf{Dom}(\beta^{\mathsf{id}}_{loc})$ and let $f \in \mathsf{Dom}(\eta_1(o))$. Note that $o_1 \notin \mathsf{Dom}(\beta^{\mathsf{id}}_{loc})$. We must prove that:
   $$\beta^{\mathsf{id}}, \lfloor \mathcal{T}_{i_1}(\beta^{\mathsf{id}^{\triangleleft-1}}(o), f) \rfloor \vdash \eta_1(o, f) \sim^\lambda \eta_1'(\beta^{\mathsf{id}}(o), f).$$
   By Lemma 3.37 (in particular, that value indistinguishability is reflexive),
   $$\beta^{\mathsf{id}}, \lfloor \mathcal{T}_{i_1}(\beta^{\mathsf{id}^{\triangleleft-1}}(o), f) \rfloor \vdash \eta_1(o, f) \sim^\lambda \eta_1(\beta^{\mathsf{id}}(o), f).$$
   Given the definition of η_1' this is equivalent to
   $$\beta^{\mathsf{id}}, \lfloor \mathcal{T}_{i_1}(\beta^{\mathsf{id}^{\triangleleft-1}}(o), f) \rfloor \vdash \eta_1(o, f) \sim^\lambda \eta_1'(\beta^{\mathsf{id}}(o), f).$$
   Finally, we resort to the hypothesis $\mathcal{T}_{i_1} \leq \mathcal{T}_{i_1+1}$ and Lemma A.1 to conclude.

Case: $B(i) = \texttt{putfield}\ f_j$
There are three options to consider:

1. There is no exception, then
   $$\frac{B(i_1) = \texttt{putfield}\ f_j \quad o_1 \in \mathsf{Dom}(\eta_1) \quad f_j \in \mathsf{Dom}(\eta_1(o_1))}{\langle i_1, \alpha_1, v_1 \cdot o_1 \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i_1+1, \alpha_1, \sigma_1, \eta_1 \oplus (\{o_1 \mapsto \eta_1(o_1)\} \oplus \{f_j \mapsto v_1\}) \rangle = s_1'}$$
   Moreover, from T-PtFld:

$$\begin{array}{l}
\mathcal{S}_{i_1} \leq_{\mathcal{A}_{i_1}} \langle R_1, l_1\rangle \cdot \langle R, l\rangle \cdot \mathcal{S}_{i+1} \\
l_1 \sqcup l \sqsubseteq \bigsqcap_{a\in R} \mathcal{T}_{i_1}(a,f) \\
for\ all\ a \in R. \mathcal{T}_{i+1}(a,f) = \langle R_3, l_2\rangle, \\
\qquad\qquad where\ \mathcal{T}_{i_1}(a,f) = \langle \_, l_2\rangle \wedge R_1 \subseteq R_3 \\
\mathcal{A}_{i_1} \sqsubseteq l, l_1 \\
\mathcal{V}_{i_1} \leq \mathcal{V}_{i+1} \\
\mathcal{T}_{i_1} \backslash \{(a,f) \mid a \in R\} \leq \mathcal{T}_{i+1} \backslash \{(a,f) \mid a \in R\} \\
\forall k \in \mathtt{region}(i). l \sqsubseteq \mathcal{A}_k
\end{array}$$

We prove all three remaining items of $\beta^{\mathsf{id}} \vdash s_1 \sim^\lambda s'_1$.

a) Idem to `load` case.

b) By Lemma A.14 (reflexivity of stack high indistinguishability), $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1} \setminus 2, \mathcal{S}_{i_1} \setminus 2), \mathcal{A}_{i_1} \vdash \sigma_1 \sim^\lambda \sigma_1$. By H-Cons-L we deduce $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1} \setminus 2), \mathcal{A}_{i_1} \vdash v_1 \cdot o_1 \cdot \sigma_1 \sim^\lambda \sigma_1$. Finally, by condition 1 of T-PtFld and Lemma A.7 we have that $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1+1}), \mathcal{A}_{i_1+1} \vdash v_1 \cdot o_1 \cdot \sigma_1 \sim^\lambda \sigma_1$.

c) We address $\beta^{\mathsf{id}}, (\mathcal{T}_{i_1}, \mathcal{T}_{i_1+1}) \vdash \eta_1 \sim^\lambda \eta'_1$.
The first item is immediate. The second item is proved as follows. Let $o \in \mathsf{Dom}(\beta^{\mathsf{id}}_{loc})$, and let $f \in \mathsf{Dom}(\eta_1(o))$. We must prove that:
$$\beta^{\mathsf{id}}, \lfloor \mathcal{T}_{i_1}(\beta^{\mathsf{id}^{\lhd-1}}(o), f)\rfloor \vdash \eta_1(o,f) \sim^\lambda \eta'_1(\beta^{\mathsf{id}}(o), f).$$
By Lemma 3.37,
$$\beta^{\mathsf{id}}, \lfloor \mathcal{T}_{i_1}(\beta^{\mathsf{id}^{\lhd-1}}(o), f)\rfloor \vdash \eta_1(o,f) \sim^\lambda \eta_1(\beta^{\mathsf{id}}(o), f).$$
We have two cases:
   i. If $\beta^{\mathsf{id}^{\lhd-1}}(o) \in \mathsf{Dom}(\mathcal{T}_{i_1}) \setminus R$, then the result follows from $\mathcal{T}_{i_1} \leq \mathcal{T}_{i_1+1}$ and Lemma A.1.
   ii. Otherwise $\beta^{\mathsf{id}^{\lhd-1}}(o) \in R$. From condition 2 and 4 of T-PtFld, $\mathcal{T}_{i_1}(\beta^{\mathsf{id}^{\lhd-1}}(o), f) \not\sqsubseteq \lambda$. Therefore,
   $$\beta^{\mathsf{id}}, \lfloor \mathcal{T}_{i_1}(\beta^{\mathsf{id}^{\lhd-1}}(o), f)\rfloor \vdash \eta_1(o,f) \sim^\lambda \eta'_1(\beta^{\mathsf{id}}(o), f)$$
   holds trivially. Note that for all other $f' \in \mathsf{Dom}(\eta_1(o))$ different from f we reason as in the previous subcase.

2. An exception is raised and there is a handler, then

$$\frac{\begin{array}{c} B(i_1) = \mathtt{putfield}\ f \quad o_1 = null \\ o' = \mathsf{Fresh}(\eta_1) \quad \mathsf{Handler}(i_1) = i' \end{array}}{\langle i_1, \alpha, v_1 \cdot o_1 \cdot \sigma_1, \eta_1\rangle \longrightarrow_B \langle i', \alpha_1, o' \cdot \epsilon, \eta_1 \oplus \{o' \mapsto \mathsf{Default}(\mathbf{Throwable})\}\rangle = s'_1}$$

Moreover, from T-PtFldH:

$$\begin{array}{l}
\mathsf{Handler}(i_1) = i' \\
\mathcal{S}_{i_1} = \kappa' \cdot \langle R, l\rangle \cdot S, \text{ for some } S \\
a = \mathsf{Fresh}(\mathcal{T}_{i_1}) \\
\langle \{a\}, l \sqcup \mathcal{A}_{i_1}\rangle \cdot \epsilon \leq_{\mathcal{A}_{i_1}} \mathcal{S}_{i'} \\
\mathcal{V}_{i_1} \leq \mathcal{V}_{i'} \\
\mathcal{T}_{i_1} \oplus \{a \mapsto [f_1 : \mathtt{ft}_{\mathcal{A}_{i_1}}(f_1), \ldots, f_n : \mathtt{ft}_{\mathcal{A}_{i_1}}(f_n)]\} \leq \mathcal{T}_{i'} \\
\forall j \in \mathsf{Dom}(\mathcal{S}_{i_1}). \mathcal{A}_{i_1} \sqsubseteq \mathcal{S}_{i_1}(j)
\end{array}$$

We prove stack indistinguishability. For the two remaining items of $\beta^{\mathsf{id}} \vdash s_1 \sim^\lambda s'_1$, we conclude as in the case of `new`.
We must prove that $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i'}), \mathcal{A}_{i'} \vdash v_1 \cdot o_1 \cdot \sigma_1 \sim^\lambda o' \cdot \epsilon$.
By L-Nil and H-Low, we can affirm that $\beta^{\mathsf{id}}, (\epsilon, \epsilon), \mathcal{A}_{i_1} \vdash \epsilon \sim^\lambda \epsilon$. Now, by condition 4 of typing rule and $\mathcal{A}_{i_1} \not\sqsubseteq \lambda$ and H-Cons-R we have that $\beta^{\mathsf{id}}, (\epsilon, \langle \_, l \sqcup \mathcal{A}_{i_1}\rangle \cdot \epsilon), \mathtt{H} \vdash \epsilon \sim^\lambda o' \cdot \epsilon$.
By the last condition of T-PtFld, we can repeatedly apply H-Cons-L for obtain $\beta', (\mathcal{S}_i, \langle \_, l \sqcup \mathcal{A}_{i_1}\rangle \cdot \epsilon), \mathcal{A}_{i_1} \vdash v_1 \cdot o_1 \cdot \sigma_1 \sim^\lambda o' \cdot \epsilon$. The result follows by $\mathcal{A}_{i'} \not\sqsubseteq \lambda$ and stack indistinguishability definition.

3. An exception is raised and there is no a handler, then

$$\frac{\begin{array}{c}B(i_1) = \texttt{putfield}\ f \quad o_1 = null \\ o_1' = \mathsf{Fresh}(\eta_1) \quad \mathsf{Handler}(i_1)\uparrow\end{array}}{\langle i_1, \alpha_1, v_1 \cdot o_1 \cdot \sigma_1, \eta_1\rangle \longrightarrow_B \langle err_i, \alpha_1, o_1' \cdot \sigma_1, \eta_1 \oplus \{o_1' \mapsto \mathsf{Default}(\mathbf{Throwable})\}\rangle = s_1'}$$

Moreover, from T-PtFldNoH:

$$\begin{array}{l}
\mathcal{S}_{i_1} \leq_{\mathcal{A}_{i_1}} \kappa \cdot \langle R, l\rangle \cdot (\mathcal{S}_{err_i}\backslash 0) \\
\kappa \sqsubseteq \bigsqcap_{a\in R} \mathcal{T}_{i_1}(a, f) \\
a = \mathsf{Fresh}(\mathcal{T}_{i_1}) \\
\langle \{a\}, l \sqcup \mathcal{A}_{i_1}\rangle \sqsubseteq \mathcal{S}_{err_i}(0) \\
l \sqsubseteq \kappa_r \\
\mathcal{A}_{i_1} \sqsubseteq \kappa, l \\
\mathcal{V}_{i_1} \leq \mathcal{V}_{err_i} \\
\mathcal{T}_{i_1} \oplus \{a \mapsto [f_1 : \texttt{ft}_{\mathcal{A}_i}(f_1), \ldots, f_n : \texttt{ft}_{\mathcal{A}_i}(f_n)]\} \leq \mathcal{T}_{err_i}
\end{array}$$

We prove all three remaining items of $\beta^{\mathsf{id}} \vdash s_1 \sim^\lambda s_1'$.

a) By Lemma 3.39 (reflexivity of variable high indistinguishability), $\beta^{\mathsf{id}}, (\mathcal{V}_{i_1}, \mathcal{V}_{i_1}), \mathcal{A}_{i_1} \vdash \alpha_1 \sim^\lambda \alpha_1$. Using this, condition 7 of T-PtFld and Lemma A.3 we conclude as in the case of T-New.

b) We address $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{err_i}), \mathcal{A}_{err_i} \vdash v_1 \cdot o_1 \cdot \sigma_1 \sim^\lambda o_1' \cdot \sigma_1$.
By Lemma A.14 (reflexivity of stack high indistinguishability) we have that $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1}), \mathcal{A}_{i_1} \vdash v_1 \cdot o_1 \cdot \sigma_1 \sim^\lambda v_1 \cdot o_1 \cdot \sigma_1$.
Now, by condition 1 of T-PtFld and Lemma A.7 we have $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \kappa \cdot \langle R, l\rangle \cdot (\mathcal{S}_{err_i}\backslash 0)), \mathcal{A}_{i_1} \vdash v_1 \cdot o_1 \cdot \sigma_1 \sim^\lambda v_1 \cdot o_1 \cdot \sigma_1$. Given that $\mathcal{A}_{i_1} \not\sqsubseteq \lambda$, by condition 6 of T-PtFld we may resort to H-Cons-R to obtain $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, (\mathcal{S}_{err_i}\backslash 0))), \mathcal{A}_{i_1} \vdash v_1 \cdot o_1 \cdot \sigma_1 \sim^\lambda \sigma_1$.
Now, by condition by condition 4 of T-PtFld and H-Cons-R, we have $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{err_i}), \mathcal{A}_{i_1} \vdash v_1 \cdot o_1 \cdot \sigma_1 \sim^\lambda o_1' \cdot \sigma_1$. The result follows by $\mathcal{A}_{err_i} \not\sqsubseteq \lambda$ and stack indistinguishability definition.

c) We conclude as in the case of `new`.

Case: $B(i_1) = \texttt{throw}$

We have two cases:

1. If there is a handler defined, then

$$\frac{B(i_1) = \texttt{throw} \quad \mathsf{Handler}(i_1) = i'}{\langle i_1, \alpha_1, o_1 \cdot \sigma_1, \eta_1\rangle \longrightarrow_B \langle i', \alpha_1, o_1 \cdot \epsilon, \eta_1\rangle = s_1'}$$

Moreover, from T-Throw:

$$\begin{array}{l}
\mathsf{Handler}(i_1) = i' \\
\mathcal{S}_{i_1}(0) \sqsubseteq \mathcal{A}(i') \\
\mathcal{V}_{i_1} \leq \mathcal{V}_{i'} \\
\mathcal{T}_{i_1} \leq \mathcal{T}_{i'} \\
\mathcal{S}_i(0) \cdot \epsilon \leq_{\mathcal{A}_{i_1}} \mathcal{S}_{i'} \\
\forall j \in \mathsf{Dom}(\mathcal{S}_{i_1}).\mathcal{A}_{i_1} \sqsubseteq \mathcal{S}_{i_1}(j) \\
\mathcal{A}(i)
\end{array}$$

We prove the three remaining items of $\beta^{\mathsf{id}} \vdash s_1 \sim^\lambda s_1'$.

a) We conclude as in the case of `load`.
b) We conclude as in the case of T-PtFldH.
c) We conclude as in the case of `store`.

2. If there is a no handler defined, then

$$\frac{B(i_1) = \texttt{throw} \quad \mathsf{Handler}(i_1)\uparrow}{\langle i_1, \alpha_1, o_1 \cdot \sigma_1, \eta_1\rangle \longrightarrow_B \langle err_i, \alpha_1, o_1 \cdot \sigma_1, \eta_1\rangle = s_1'}$$

Moreover, from T-Throw:

$$\begin{array}{l} \mathsf{Handler}(i_1)\uparrow \\ \mathcal{S}_{i_1}(0) \sqcup \mathcal{A}_{i_1} \sqsubseteq \kappa_r \\ \mathcal{S}_{i_1} \leq_{\mathcal{A}_{i_1}} \mathcal{S}_{err_i} \\ \mathcal{V}_{i_1} \leq \mathcal{V}_{err_i} \\ \mathcal{T}_{i_1} \leq \mathcal{T}_{err_i} \end{array}$$

We prove stack indistinguishability. For the two remaining items of $\beta^{\mathsf{id}} \vdash s_1 \sim^\lambda s_1'$, we conclude as in the previous case.
By Lemma A.14 (reflexivity of stack high indistinguishability) we have that $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1}), \mathcal{A}_{i_1} \vdash o_1 \cdot \sigma_1 \sim^\lambda o_1 \cdot \sigma_1$. Finally, by condition 3 of T-Throw and Lemma A.7 we have $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{err_i}), \mathcal{A}_{i_1} \vdash o_1 \cdot \sigma_1 \sim^\lambda o_1 \cdot \sigma_1$. Now, $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{err_i}), \mathcal{A}_{err_i} \vdash o_1 \cdot \sigma_1 \sim^\lambda o_1 \cdot \sigma_1$, follow by stack indistinguishability and $\mathcal{A}_{err_i} \not\sqsubseteq \lambda$.

Case: $B(i_1) = \texttt{getfield}\ f_j$
There are three options to consider:

1. There is no exception, then
   - By the operational semantics we know:
   $$\frac{B(i_1) = \texttt{getfield}\ f \quad o_1 \in \mathsf{Dom}(\eta_1) \quad \eta_1(o_1, f) = v_1}{\langle i_1, \alpha_1, o_1 \cdot \sigma_1, \eta_1\rangle \rightarrow_B \langle i_1+1, \alpha_1, v_1 \cdot \sigma_1, \eta_1\rangle}$$
   i.e. $s_1' = \langle i_1+1, \alpha_1, v_1 \cdot \sigma_1, \eta_1\rangle$.
   - From T-GtFld:

$$\begin{array}{l} \mathsf{Handler}(i_1)\downarrow \\ \mathcal{S}_{i_1}(0) = \langle R, l\rangle \\ \kappa = \bigsqcup_{a \in R} \mathcal{T}_{i_1}(a, f) \\ l \sqcup \kappa \sqcup \mathcal{A}_{i_1} \cdot (\mathcal{S}_{i_1} \backslash 0) \leq_{\mathcal{A}_{i_1}} \mathcal{S}_{i_1+1} \\ \mathcal{V}_{i_1} \leq \mathcal{V}_{i_1+1} \\ \mathcal{T}_{i_1} \leq \mathcal{T}_{i_1+1} \\ \forall k \in \texttt{region}(i_1). l \sqsubseteq \mathcal{A}_k \end{array}$$

In order to prove $\beta^{\mathsf{id}} \vdash s_1 \sim^\lambda s_1'$, the only case of interest is stack indistinguishability.
By Lemma A.14 (reflexivity of stack high indistinguishability), $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1} \setminus 0, \mathcal{S}_{i_1} \setminus 0), \mathcal{A}_{i_1} \vdash \sigma_1 \sim^\lambda \sigma_1$. Then by H-Cons-L and condition 4, $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1} \setminus 0), \mathcal{A}_{i_1} \vdash o_1 \cdot \sigma_1 \sim^\lambda \sigma_1$. Now, by H-Cons-R,$\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, l \sqcup l \cdot \mathcal{S}_{i_1} \setminus 0), \mathcal{A}_{i_1} \vdash o_1 \cdot \sigma_1 \sim^\lambda v_1 \cdot \sigma_1$.
Finally, by condition 4 of T-GtFld and Lemma A.7 we conclude with $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1+1}), \mathcal{A}_{i_1} \vdash o_1 \cdot \sigma_1 \sim^\lambda v_1 \cdot \sigma_1$. The result follows by stack indistinguishability and $\mathcal{A}_{i+1} \not\sqsubseteq \lambda$.
2. An exception is raised and there is a handler.

3. An exception is raised and there is no a handler.
   The proof the both cases it is the same way that the case for `putfield`.

Cases: We assume $B(i_1) = \texttt{return}$.
By the operational semantics we know:

$$\frac{B(i_1) = \texttt{return}}{\langle i_1, \alpha_1, v_1 \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle p_f, v_1, \eta_1 \rangle = s'_1}$$

From T-Ret:

$$\begin{array}{l} \mathcal{S}_{i_1}(0) \sqcup \mathcal{A}_{i_1} \sqsubseteq \kappa_r \\ \mathcal{T}_{i_1} \leq \mathcal{T}_{p_f} \end{array}$$

The $\langle i_1, v_1, \eta_1 \rangle \sim^{\lambda}_{\beta^{\mathsf{id}}} \langle p_f, v_1, \eta_1 \rangle$ is trivial.
Cases: We assume $\texttt{pc}(i_1) = err_{i_1}$.
By the operational semantics we know:

$$\frac{B(err_{i_1})}{\langle err_{i_1}, \alpha_1, v_1 \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle p_f, v_1, \eta_1 \rangle = s'_1}$$

From T-Ret:

$$\begin{array}{l} \mathcal{S}_{err_{i_1}}(0) \sqcup \mathcal{A}(err_{i_1}) \sqsubseteq \kappa_r \\ \mathcal{T}_{err_{i_1}} \leq \mathcal{T}_{p_f} \end{array}$$

The $\langle err_{i_1}, v_1, \eta_1 \rangle \sim^{\lambda}_{\beta^{\mathsf{id}}} \langle p_f, v_1, \eta_1 \rangle$ is trivial.

Case: $B(i) = \texttt{invokevirtual}\ B'$
There are three options to consider:

1. There is no exception, then

$$\frac{\begin{array}{c} B(i) = \texttt{invokevirtual}\ m \quad o_1 \in \mathsf{Dom}(\eta_1) \quad B' = \mathsf{lookup}(m, \mathsf{dynamic}(o_1)) \\ \langle 1, \{this \mapsto o_1, \overrightarrow{x_1 \mapsto v_1}\}, \epsilon, \eta_1 \rangle \longrightarrow^*_{B'} \langle p'_f, v_1, \eta'_1 \rangle \end{array}}{\langle i_1, \alpha_1, \overrightarrow{v_1} \cdot o_1 \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i_1 + 1, \alpha_1, v_1 \cdot \sigma_1, \eta'_1 \rangle = s'_1}$$

From T-InvkVrtl:

$$\begin{array}{l} \mathsf{mt}_m(\kappa \sqcup \mathcal{A}_{i_1}) = \mathcal{A}'_1, \mathcal{T}'_1, \mathcal{T}'_{p_f} \rhd ((\overrightarrow{x' : \kappa'}, \kappa'_r, E'), B') \\ \mathcal{A}_{i_1} \sqcup \kappa = \mathcal{A}'_1 \\ \mathcal{S}_{i_1} = \kappa'_1 \cdots \kappa'_n \cdot \kappa \cdot S, \text{ for some } S \\ \mathcal{S}_{i_1} \leq_{\mathcal{A}_{i_1}} \kappa'_1 \cdots \kappa'_n \cdot \kappa \cdot (\mathcal{S}_{i_1+1} \backslash 0) \\ \mathcal{A}_{i_1} \sqsubseteq \kappa'_1, \cdots, \kappa'_n, \kappa \\ \kappa \sqcup \kappa'_r \sqcup \mathcal{A}_{i_1} \sqsubseteq \mathcal{S}_{i_1+1}(0) \\ \mathcal{V}_{i_1} \leq \mathcal{V}_{i_1+1} \\ \mathcal{T}_{i_1} \leq \mathcal{T}'_1 \leq \mathcal{T}'_{p_f} \leq \mathcal{T}_{i_1+1} \\ \forall k \in \texttt{region}(i_1).\kappa \sqcup \kappa'_r \sqsubseteq \mathcal{A}_k, \text{ if } i_1 \in \mathsf{Dom}(B^\sharp) \end{array}$$

In order to prove $\beta^{\mathsf{id}} \vdash s_1 \sim^{\lambda} s_1'$, the only case of interest is stack indistinguishability. By Lemma A.14 (reflexivity of stack high indistinguishability), $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1} \setminus \|\kappa_1' \cdots \kappa_n' \cdot \kappa\|, \mathcal{S}_{i_1} \setminus \|\kappa_1' \cdots \kappa_n' \cdot \kappa\|), \mathcal{A}_{i_1} \vdash \sigma_1 \sim^{\lambda} \sigma_1$. By H-Cons-L we deduce $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1} \setminus \|\kappa_1' \cdots \kappa_n' \cdot \kappa\|), \mathcal{A}_{i_1} \vdash \overrightarrow{v_1} \cdot o_1 \cdot \sigma_1 \sim^{\lambda} \sigma_1$. Now, by condition 4 of T-InvkVrtl and Lemma A.7 we have that $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1+1} \setminus 0), \mathcal{A}_{i_1} \vdash \overrightarrow{v_1} \cdot o_1 \cdot \sigma_1 \sim^{\lambda} \sigma_1$. Finally, by condition 5 y 6 of T-InvkVrtl and H-Cons-L we have that $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1+1}), \mathcal{A}_{i_1} \vdash \overrightarrow{v_1} \cdot o_1 \cdot \sigma_1 \sim^{\lambda} v_1 \cdot \sigma_1$. The result follows by stack indistinguishability and $\mathcal{A}_{i_1+1} \not\sqsubseteq \lambda$.

2. An exception is raised and there is a handler.
3. An exception is raised and there is no a handler.

The proof the both cases it is the same way that the case for `putfield`.

Case: $B(i) = \texttt{goto}\ i'$ is trivial by that it does not change the state of the program.

Case: $B(i) = \texttt{push}\ v$ and $B(i) = \texttt{prim}$ are similar to case $B(i) = \texttt{load}\ x$.

Case: $B(i) = \texttt{pop}$ is similar to case $B(i) = \texttt{store}\ x$.

Case: $B(i) = \texttt{if}\ i'$ the four cases are similar to case $B(i) = \texttt{store}\ x$.

∎

A.4 High-Low Context

The proof of the Equivalence in High-Low Level Contexts Lemma follows. The only interesting cases are `goto`, `return` and $\texttt{pc}(s_1)=err_i$.

Lemma 3.46 (One-Step High to Low)
Let $i_1, i_2 \in \mathsf{region}(k)$ for some k such that $\forall k' \in \mathsf{region}(k) : \mathcal{A}_{k'} \not\sqsubseteq \lambda$. Suppose:

1. $s_1 \longrightarrow_B s_1'$ and $s_2 \longrightarrow_B s_2'$;
2. $\beta \vdash s_1 \sim^{\lambda} s_2$, for some location bijection set β; and
3. $s_1' = \langle i_1', \alpha_1', \sigma_1', \eta_1' \rangle$ and $s_2' = \langle i_2', \alpha_2', \sigma_2', \eta_2' \rangle$;
4. and $\mathcal{A}_{i_1'} = \mathcal{A}_{i_2'}$ and $\mathcal{A}_{i_1'} \sqsubseteq \lambda$.

Then $i_1' = \mathsf{jun}(k) = i_2'$ and $\beta \vdash s_1' \sim^{\lambda} s_2'$.

Proof. There are several options to consider.

- Due to our assumption on the forms of programs (see comment below Property 3.10), we may assume both $B(i_1) = \texttt{goto}\ i_1'$ and $B(i_2) = \texttt{goto}\ i_2'$. Thus, $s_1' = \langle i_1', \alpha_1, \sigma_1, \eta_1 \rangle$ and $s_2' = \langle i_2', \alpha_2, \sigma_2, \eta_2 \rangle$ and both of the following hold:

$$\frac{B(i_1) = \texttt{goto}\ i_1'}{\langle i_1, \alpha_1, \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i_1', \alpha_1, \sigma_1, \eta_1 \rangle} \qquad \frac{B(i_2) = \texttt{goto}\ i_2'}{\langle i_2, \alpha_2, \sigma_2, \eta_2 \rangle \longrightarrow_B \langle i_2', \alpha_2, \sigma_2, \eta_2 \rangle}$$

Also, from T-Goto:

$$\begin{array}{ll} \mathcal{V}_{i_1} \leq \mathcal{V}_{i_1'} & \mathcal{V}_{i_2} \leq \mathcal{V}_{i_2'} \\ \mathcal{S}_{i_1} \leq_{\mathcal{A}_{i_1}} \mathcal{S}_{i_1'} & \mathcal{S}_{i_2} \leq_{\mathcal{A}_{i_2}} \mathcal{S}_{i_2'} \\ \mathcal{T}_{i_1} \leq \mathcal{T}_{i_1'} & \mathcal{T}_{i_2} \leq \mathcal{T}_{i_2'} \end{array}$$

We now address the proof of both items:

I- $i'_1 = \mathsf{jun}(k) = i'_2$. By the hypothesis $\mathcal{A}_{i_1}, \mathcal{A}_{i_2} \not\sqsubseteq \lambda$, $i_1 \mapsto i'_1$, $i_2 \mapsto i'_2$ and by assumption $\mathcal{A}_{i'_1} = \mathcal{A}(i'_2) \sqsubseteq \lambda$. Therefore, by Lemma A.12 we know that $i'_1 = \mathsf{jun}(k)$ and that $i'_2 = \mathsf{jun}(k)$. But by the SOAP property, junction points are unique. Hence $i'_1 = \mathsf{jun}(k) = i'_2$.

II- We address the proof of the four items of the definition of state indistinguishability for $\beta \vdash s'_1 \sim^\lambda s'_2$:

1. By hypothesis $\mathcal{A}_{i'_1} = \mathcal{A}_{i'_2}$ and $\mathcal{A}_{i'_1} \sqsubseteq \lambda$.
2. We must prove $\beta, \mathcal{V}_{i'_1}, \mathcal{A}_{i'_1} \vdash \alpha_1 \sim^\lambda \alpha_2$ (since $\alpha'_1 = \alpha_1$ and $\alpha'_2 = \alpha_2$). Recall from above that by hypothesis:
$$\beta, (\mathcal{V}_{i_1}, \mathcal{V}_{i_2}), l \vdash \alpha_1 \sim^\lambda \alpha_2, l \not\sqsubseteq \lambda \tag{A.21}$$
Let $x \in \mathbb{X}$. We have two cases:
   - If $\mathcal{V}_{i'_1}(x) \not\sqsubseteq \lambda$, then we trivially have $\beta, \lfloor \mathcal{V}_{i'_1}(x) \rfloor \vdash \alpha_1(x) \sim^\lambda \alpha_2(x)$.
   - If $\mathcal{V}_{i'_1}(x) \sqsubseteq \lambda$, then $\mathcal{V}_{i_1}(x) \sqsubseteq \lambda$ and $\mathcal{V}_{i_2}(x) \sqsubseteq \lambda$. Thus $\beta, \lfloor \mathcal{V}_{i'_1}(x) \rfloor \vdash \alpha_1(x) \sim^\lambda \alpha_2(x)$ follows from (A.21).
3. We must prove $\beta, \mathcal{S}_{i'_1}, \mathcal{A}_{i'_1} \vdash \sigma_1 \sim^\lambda \sigma_2$. By the hypothesis,
$$\beta, (\mathcal{S}_{i_1}, \mathcal{S}_{i_2}), l \vdash \sigma_1 \sim^\lambda \sigma_2, l \not\sqsubseteq \lambda \tag{A.22}$$
and by condition 2 of T-Goto we know that $\|\mathcal{S}_{i_1}\| = \|\mathcal{S}_{i'_1}\| = \|\mathcal{S}_{i_2}\|$ (also, $\mathcal{S}_{i'_1} = \mathcal{S}_{i'_2}$).
For each $j \in \{0..\|\mathcal{S}_{i'_1}\| - 1\}$, we proceed as follows:
   - If $\mathcal{S}_{i'_1}(j) \not\sqsubseteq \lambda$, then trivially $\beta, \lfloor \mathcal{S}_{i'_1}(j) \rfloor \vdash \sigma_1(j) \sim^\lambda \sigma_2(j)$.
   - If $\mathcal{S}_{i'_1}(j) \sqsubseteq \lambda$, then $\mathcal{S}_{i_1}(j) = \mathcal{S}_{i_2}(j) \sqsubseteq \lambda$. From (A.22) and the definition of stack indistinguishability, it follows that there exists $k \geq j$ such that the prefix of size k of σ_1 and σ_2 are low indistinguishable. As a consequence, $\beta, \lfloor \mathcal{S}_{i'_1}(j) \rfloor \vdash \sigma_1(j) \sim^\lambda \sigma_2(j)$.

   Hence, $\beta, \mathcal{S}_{i'_1}, \mathcal{A}_{i'_1} \vdash \sigma_1 \sim^\lambda \sigma_2$.
4. We address the two items of the definition of $\beta, \mathcal{T}_{i'_1} \vdash \eta_1 \sim^\lambda \eta_2$. The first follow from the hypothesis that s_1 and s_2 are indistinguishable and $\mathcal{T}_{i_1}, \mathcal{T}_{i_2} \leq \mathcal{T}_{i'_1}$. Regarding the last item, we proceed as follows. Let $o \in \mathsf{Dom}(\beta_{loc})$, and let $f \in \mathsf{Dom}(\eta_1(o))$. We must prove that:
$$\beta, \lfloor \mathcal{T}_{i'_1}(\beta^{\lhd -1}(o), f) \rfloor \vdash \eta_1(o, f) \sim^\lambda \eta_2(\beta(o), f).$$
This is an immediate consequence of the hypothesis, $\sim^\lambda_\beta \mathcal{T}_{i_1}(\beta^{\lhd -1}(o), f) \eta_1(o, f) \eta_2(\beta(o), f)$ and $\mathcal{T}_{i_1}, \mathcal{T}_{i_2} \leq \mathcal{T}_{i'_1}$.

- We may assume both $B(i_1) = \texttt{return}$ and $B(i_2) = \texttt{return}$. Thus, $s'_1 = \langle p_f, v_1, \eta_1 \rangle$ and $s'_2 = \langle p_f, v_2, \eta_2 \rangle$ and both of the following hold:

$$\frac{B(i_1) = \texttt{return}}{\langle i_1, \alpha_1, v_1 \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle p_f, v_1, \eta_1 \rangle} \qquad \frac{B(i_2) = \texttt{return}}{\langle i_2, \alpha_2, v_2 \cdot \sigma_2, \eta_2 \rangle \longrightarrow_B \langle p_f, v_2, \eta_2 \rangle}$$

Also, from T-Ret:

$$\begin{array}{ll} \mathcal{S}_{i_1}(0) \sqcup \mathcal{A}_{i_1} \sqsubseteq \kappa_r & \quad \mathcal{S}_{i_2}(0) \sqcup \mathcal{A}_{i_2} \sqsubseteq \kappa_r \\ \mathcal{T}_{i_1} \leq \mathcal{T}_{p_f} & \quad \mathcal{T}_{i_2} \leq \mathcal{T}_{p_f} \end{array}$$

We now address the proof of both items:

I- By successor relation definition $\texttt{pc}(s'_1) = \texttt{pc}(s'_2) = p_f$.

II- We address the proof of the two items of the definition of final state indistinguishability for $\beta \vdash s'_1 \sim^\lambda s'_2$:

1. By hypothesis $\mathcal{A}(p_f) \sqsubseteq \lambda$.
2. We must prove $\beta, \lfloor \kappa_r \rfloor \vdash v_1 \sim^\lambda v_2$. By the hypothesis, $\beta, (\mathcal{S}_{i_1}, \mathcal{S}_{i_2}), l \vdash v_1 \cdot \sigma_1 \sim^\lambda v_2 \cdot \sigma_2$, $l \not\sqsubseteq \lambda$, then we have $\beta, \lfloor \mathcal{S}_{i_1}(0) \sqcup \mathcal{S}_{i_2}(0) \rfloor \vdash v_1 \sim^\lambda v_2$.
Furthermore, by T-Ret $\mathcal{S}_{i_1}(0) \sqsubseteq \kappa_r$ and $\mathcal{S}_{i_2}(0) \sqsubseteq \kappa_r$.
Now, by indistinguishability definition $\beta, \lfloor \kappa_r \rfloor \vdash v_1 \sim^\lambda v_2$.
3. Idem to previous `T-Goto` case.

- The remain cases $B(i_1) = \texttt{return}$ and $\texttt{pc}(i_2) = err_i$, $\texttt{pc}(i_1) = err_i$, $B(i_2) = \texttt{return}$ and $\texttt{pc}(i_1) = err_i$, $\texttt{pc}(i_2) = err_i$ are similar to the previous case.

■

B

Robustness Soundness Proofs

In this appendix we present detailed proofs of the lemmas introduced in Chapter 4.

B.1 Integrity Indistinguishability - Properties

Determining that integrity indistinguishability of values, local variable arrays, stacks and heaps is an equivalence relation requires the same careful consideration that for indistinguishability properties of Section 3.6.2. The proofs are similar at proof in Section A.1.1.

Lemma B.1.

1. *$\beta^{\mathsf{id}}, l \vdash v \simeq^{\mathtt{A}} v$, if $v \in \mathbb{L}$ and $l \in \mathtt{H}_I$ implies $v \in \mathsf{Dom}(\beta^{\mathsf{id}})$.*
2. *$\beta, l \vdash v_1 \simeq^{\mathtt{A}} v_2$ implies $\hat{\beta}, l \vdash v_2 \simeq^{\mathtt{A}} v_1$.*
3. *If $\beta, l \vdash v_1 \simeq^{\mathtt{A}} v_2$ and $\gamma, l \vdash v_2 \simeq^{\mathtt{A}} v_3$, then $\gamma \circ \beta, l \vdash v_1 \simeq^{\mathtt{A}} v_3$.*

Proof. The first two items follow directly from a close inspection of the definition of value indistinguishability. Regarding transitivity we consider the two cases, $l \notin \mathtt{H}_I$ and $l \in \mathtt{H}_I$. In the former case, $\gamma \circ \beta, l \vdash v_1 \simeq^{\mathtt{A}} v_3$ holds trivially by definition of value indistinguishability. For the latter, if $v_1 = null$ or $v_1 \in \mathbb{Z}$ we resort to transitivity of equality. If $v_1 \in \mathbb{L}$, then by hypothesis and by the definition of value indistinguishability, $v_2, v_3 \in \mathbb{L}$, $\beta(v_1) = v_2$ and $\gamma(v_2) = v_3$. And by definition of $\gamma \circ \beta$, $(\gamma \circ \beta)(v_1) = \gamma(\beta(v_1)) = v_3$. Hence $\gamma \circ \beta, l \vdash v_1 \simeq^{\mathtt{A}} v_3$. ∎

We now address local variable arrays. Variable reuse allows a public variable to be reused for storing secret information in a high security execution context. Suppose, therefore, that $l \notin \mathtt{H}_I$, $\beta, (V_1, V_2), l \vdash \alpha_1 \simeq^{\mathtt{A}} \alpha_2$ and $\gamma, (V_2, V_3), l \vdash \alpha_2 \simeq^{\mathtt{A}} \alpha_3$ where, for some x, $V_1(x) = \mathtt{L}$, $V_2(x) = \mathtt{H}$ and $V_3(x) = \mathtt{L}$. Clearly it is not necessarily the case that $\gamma \circ \beta, (V_1, V_3), l \vdash \alpha_1 \simeq^{\mathtt{A}} \alpha_3$ given that $\alpha_1(x)$ and $\alpha_3(x)$ may differ. We thus require that either V_1 or V_3 have at least the level of V_2 for this variable: $V_2(x) \sqsubseteq V_1(x)$ or $V_2(x) \sqsubseteq V_3(x)$. Of course, it remains to be seen that such a condition can be met when proving noninterference, we defer that discussion to Sec. 3.6.3. For now we state the result, namely that indistinguishability of local variable arrays is an equivalence relation (its proof is an easy consequence of lemma 3.37).

Notation B.2

- $\mathsf{HighLoc}(\alpha, V, \mathtt{A})$ (or simply $\mathsf{HighLoc}(\alpha)$ if the V is understood from the context) is shorthand for $\{o \mid \alpha(x) = o, V(x) \in \mathtt{H}_I\}$.
- $\mathsf{HighLoc}(\sigma, S, \mathtt{A})$ (or simply $\mathsf{HighLoc}(\sigma)$ if the S is understood from the context) is defined as $\{o \mid \exists i \in 0..\|S\| - 1. \sigma(i) = o, S(i) \in \mathtt{H}_I\}$
- $\mathsf{HighLoc}(\eta, \beta, T, \mathtt{A})$ (or simply $\mathsf{HighLoc}(\eta, \beta)$ if the T is understood from the context) is defined as $\{o' \mid o \in \mathsf{Dom}(\beta), \exists f \in \mathsf{Dom}(\eta(o)).(\eta(o, f) = o', T(\beta^{-1}_{sloc}(o), f) \in \mathtt{H}_I\}$

- $\mathsf{HighLoc}(\langle i,\alpha,\sigma,\eta\rangle,\beta,\langle i,V,S,T\rangle,\mathtt{A})$ (or simply $\mathsf{HighLoc}(s,\beta,\mathtt{A})$ if the $\langle i,V,S,T\rangle$ is understood from the context) is defined as $\mathsf{HighLoc}(\alpha,V,\mathtt{A})\cup\mathsf{HighLoc}(\sigma,S,\mathtt{A})\cup\mathsf{HighLoc}(\eta,\beta,T,\mathtt{A})$

Lemma B.3. *For low indistinguishability we have ($l\in\mathtt{H}_I$):*

1. $\beta^{\mathsf{id}},V,l\vdash\alpha\simeq^{\mathtt{A}}\alpha$, *if* $\mathit{HighLoc}(\alpha,V,\mathtt{A})\subseteq\mathsf{Dom}(\beta^{\mathsf{id}})$.
2. $\beta,V,l\vdash\alpha_1\simeq^{\mathtt{A}}\alpha_2$ *implies* $\hat{\beta},V,l\vdash\alpha_2\simeq^{\mathtt{A}}\alpha_1$.
3. *If* $\beta,V,l\vdash\alpha_1\simeq^{\mathtt{A}}\alpha_2$, $\gamma,V,l\vdash\alpha_2\simeq^{\mathtt{A}}\alpha_3$, *then* $\gamma\circ\beta,V,l\vdash\alpha_1\simeq^{\mathtt{A}}\alpha_3$.

For high indistinguishability ($l\notin\mathtt{H}_I$) we have:

1. $\beta^{\mathsf{id}},(V,V),l\vdash\alpha\simeq^{\mathtt{A}}\alpha$, *if* $\mathit{HighLoc}(\alpha,V,\mathtt{A})\subseteq\mathsf{Dom}(\beta^{\mathsf{id}})$.
2. $\beta,(V_1,V_2),l\vdash\alpha_1\simeq^{\mathtt{A}}\alpha_2$ *implies* $\hat{\beta},(V_2,V_1),l\vdash\alpha_2\simeq^{\mathtt{A}}\alpha_1$.
3. *If* $\beta,(V_1,V_2),l\vdash\alpha_1\simeq^{\mathtt{A}}\alpha_2$, $\gamma,(V_2,V_3),l\vdash\alpha_2\simeq^{\mathtt{A}}\alpha_3$ *and* $\forall x\in\mathbb{X}.V_2(x)\sqsubseteq V_1(x)$ *or* $V_2(x)\sqsubseteq V_3(x)$, *then* $\gamma\circ\beta,(V_1,V_3),l\vdash\alpha_1\simeq^{\mathtt{A}}\alpha_3$.

The condition on β^{id} simply ensures that it is defined on the appropriate locations. This too is a condition that shall always be met given that in the proof of noninterference id is always taken to be the domain of the appropriate heap.

The case of stacks and heaps are dealt with similarly. Together these results determine that machine state indistinguishability too is an equivalence relation.

Lemma B.4. *The stack indistinguishability relation is an equivalence relation.*

1. $\beta^{\mathsf{id}},(S,S),l\vdash\sigma\simeq^{\lambda}\sigma$, *if* $\mathit{HighLoc}(\alpha,S,\lambda)\subseteq\mathsf{Dom}(\beta^{\mathsf{id}}_{loc})$.
2. $\beta,(S_1,S_2),l\vdash\sigma_1\simeq^{\lambda}\sigma_2$ *implies* $\hat{\beta},(S_2,S_1),l\vdash\sigma_2\simeq^{\lambda}\sigma_1$.
3. *If* $\beta,(S_1,S_2),l\vdash\sigma_1\simeq^{\lambda}\sigma_2$ *and* $\gamma,(S_2,S_3),l\vdash\sigma_2\simeq^{\lambda}\sigma_3$, *then* $\gamma\circ\beta,(S_1,S_3),l\vdash\sigma_1\simeq^{\lambda}\sigma_3$.

Lemma B.5. *Heap indistinguishability is an equivalence relation.*

1. $\beta^{\mathsf{id}},(T,T)\vdash\eta\simeq^{\lambda}\eta$, *where* $\mathsf{Dom}(\beta^{\mathsf{id}}_{loc})\subseteq\mathsf{Dom}(\eta)$, $\mathsf{Dom}(\beta^{\mathsf{id}\triangleleft}),\mathsf{Dom}(\beta^{\mathsf{id}\triangleright})\subseteq\mathsf{Dom}(T)$ *and* $\mathit{HighLoc}(\eta,\beta^{\mathsf{id}},T,\lambda)\subseteq\mathsf{Dom}(\beta^{\mathsf{id}}_{loc})$.
2. $\beta,(T_1,T_2)\vdash\eta_1\simeq^{\lambda}\eta_2$ *implies* $\hat{\beta},(T_2,T_1)\vdash\eta_2\simeq^{\lambda}\eta_1$ *assuming for every* $o\in\mathsf{Dom}(\beta)$, $f\in\eta_1(o,f)$, $T_1(\beta^{\triangleleft-1}(o),f)=T_2(\beta^{\triangleright-1}(\beta(o)),f)$.
3. *If* $\beta,(T_1,T_2)\vdash\eta_1\simeq^{\lambda}\eta_2$, $\gamma,(T_2,T_3)\vdash\eta_2\simeq^{\lambda}\eta_3$ *then* $\gamma\circ\beta,(T_1,T_3)\vdash\eta_1\simeq^{\lambda}\eta_3$.

Lemma B.6. *For low indistinguishability we have:*

1. $\beta^{\mathsf{id}}\vdash s\simeq^{\mathtt{A}}s$, *where* $\mathit{HighLoc}(s,\beta,\mathtt{A})\subseteq\mathsf{Dom}(\beta^{\mathsf{id}})$ *and* $\mathsf{Ran}(\beta^{\mathsf{id}\triangleleft}),\mathsf{Ran}(\beta^{\mathsf{id}\triangleright})\subseteq\mathsf{Dom}(\eta)$ *and* $\mathsf{Dom}(\beta^{\mathsf{id}\triangleleft}),\mathsf{Dom}(\beta^{\mathsf{id}\triangleright})\subseteq\mathsf{Dom}(T)$.
2. $\beta\vdash s_1\simeq^{\mathtt{A}}s_2$ *implies* $\hat{\beta}\vdash s_2\simeq^{\mathtt{A}}s_1$.
3. *If* $\beta\vdash s_1\simeq^{\mathtt{A}}s_2$, $\gamma\vdash s_2\simeq^{\mathtt{A}}s_3$, *then* $\gamma\circ\beta\vdash s_1\simeq^{\mathtt{A}}s_3$.

For high indistinguishability we have:

1. $\beta^{\mathsf{id}}\vdash s\simeq^{\mathtt{A}}s$, *where* $\mathit{HighLoc}(s,\beta,\mathtt{A})\subseteq\mathsf{Dom}(\beta^{\mathsf{id}})$ *and* $\mathsf{Ran}(\beta^{\mathsf{id}\triangleleft}),\mathsf{Ran}(\beta^{\mathsf{id}\triangleright})\subseteq\mathsf{Dom}(\eta)$ *and* $\mathsf{Dom}(\beta^{\mathsf{id}\triangleleft}),\mathsf{Dom}(\beta^{\mathsf{id}\triangleright})\subseteq\mathsf{Dom}(T)$.
2. $\beta\vdash s_1\simeq^{\mathtt{A}}s_2$ *implies* $\hat{\beta}\vdash s_2\simeq^{\mathtt{A}}s_1$.
3. *Suppose* $\beta\vdash s_1\simeq^{\mathtt{A}}s_2$ *and* $\gamma\vdash s_2\simeq^{\mathtt{A}}s_3$. *Furthermore, assume* $\forall x\in\mathbb{X}.V_2(x)\sqsubseteq V_1(x)$ *or* $V_2(x)\sqsubseteq V_3(x)$. *Then* $\gamma\circ\beta\vdash s_1\simeq^{\mathtt{A}}s_3$.

B.2 Preliminaries

These auxiliary lemmas and their proofs are similar to those presented in the Appendix A.

Lemma B.7. *$\beta, l \vdash v_1 \simeq^{\lambda} v_2$ and $\beta \subseteq \beta'$ and $l' \in \mathtt{H}_I$ implies $\beta', l' \vdash v_1 \simeq^{\lambda} v_2$.*

Lemma B.8. *If $\beta, V, l \vdash \alpha_1 \simeq^{\lambda} \alpha_2$, $\beta \subseteq \beta'$, $l \in \mathtt{H}_I$ and $V \leq V'$, then $\beta', V', l \vdash \alpha_1 \simeq^{\lambda} \alpha_2$.*

Lemma B.9. *If $\beta, (V, V'), l \vdash \alpha_1 \simeq^{\lambda} \alpha_2$, $l \notin \mathtt{H}_I$ and $V' \leq V''$ then $\beta, (V, V''), l \vdash \alpha_1 \simeq^{\lambda} \alpha_2$.*

Lemma B.10. *If $\beta, (V', V), l \vdash \alpha_1 \simeq^{\lambda} \alpha_2$, $l \notin \mathtt{H}_I$ and $V' \leq V''$ then $\beta, (V'', V), l \vdash \alpha_1 \simeq^{\lambda} \alpha_2$.*

Proof. Immediate consequence of lemma B.7. ▪

Lemma B.11. *If $\beta, V, l \vdash \alpha_1 \simeq^{\lambda} \alpha_2$, $l \in \mathtt{H}_I$ and $l' \in \mathtt{H}_I$ then $\beta, V, l' \vdash \alpha_1 \simeq^{\lambda} \alpha_2$.*

Proof. We have, by hypothesis, that for all $x \in \mathbb{X} : \beta, \lfloor V(x) \rfloor \vdash \alpha_1(x) \simeq^{\lambda} \alpha_2(x)$ holds. Let $l' \in \mathtt{H}_I$ then follows $\beta, V, l' \vdash \alpha_1 \simeq^{\lambda} \alpha_2$ by local variable assignment indistinguishability definition. ▪

Lemma B.12. *If $\beta, S, l \vdash \sigma_1 \simeq^{\lambda} \sigma_2$, $\beta \subseteq \beta'$, $l \in \mathtt{H}_I$ and $S \leq_l S'$ then $\beta', S', l \vdash \sigma_1 \simeq^{\lambda} \sigma_2$.*

Lemma B.13. *If $\beta, (S, S'), l \vdash \sigma_1 \simeq^{\lambda} \sigma_2$, $l \notin \mathtt{H}_I$ and $S' \leq_l S''$, then $\beta, (S, S''), l \vdash \sigma_1 \simeq^{\lambda} \sigma_2$.*

Lemma B.14. *If $\beta, (S', S), l \vdash \sigma_1 \simeq^{\lambda} \sigma_2$, $l \notin \mathtt{H}_I$ and $S' \leq_l S''$, then $\beta, (S'', S), l \vdash \sigma_1 \simeq^{\lambda} \sigma_2$.*

Lemma B.15. *If $\beta, S, l \vdash \sigma_1 \simeq^{\lambda} \sigma_2$, $l \in \mathtt{H}_I$ and $l' \in \mathtt{H}_I$ then $\beta, S, l' \vdash \sigma_1 \simeq^{\lambda} \sigma_2$.*

Proof. We proceed by induction on the the derivation of $\beta, S, l \vdash \sigma_1 \simeq^{\lambda} \sigma_2$. The base case is straightforward given that $\beta, \epsilon, l' \vdash \epsilon \simeq^{\lambda} \epsilon$ follows from $l' \in \mathtt{H}_I$ and L-Nil-I. For the inductive case, we assume that $\beta, l'' \cdot S, l \vdash v_1 \cdot \sigma_1' \simeq^{\lambda} v_2 \cdot \sigma_2'$ is derivable and that the derivation ends in an application of Cons-L-I. We must prove that $\beta, l'' \cdot S, l' \vdash v_1 \cdot \sigma_1' \simeq^{\lambda} v_2 \cdot \sigma_2'$. By Cons-L-I we know that $\beta, l'' \vdash v_1 \simeq^{\lambda} v_2$ (1) and $\beta, S, l \vdash \sigma_1 \simeq^{\lambda} \sigma_2$. Now, by the I.H. we have that $\beta, S, l' \vdash \sigma_1 \simeq^{\lambda} \sigma_2$ (2). Then by (1), (2) and Cons-L-I $\beta, l'' \cdot S, l' \vdash v_1 \cdot \sigma_1 \simeq^{\lambda} v_2 \cdot \sigma_2$ holds. ▪

Lemma B.16. *$\beta, (T, T) \vdash \eta_1 \simeq^{\lambda} \eta_2$ and $T \leq T'$ and $T \leq T''$ and $\beta \subseteq \beta'$ such that*

1. *$\beta'_{loc} = \beta_{loc}$,*
2. *$\mathsf{Ran}(\beta'^{\triangleleft}) \subseteq \mathsf{Dom}(\eta_1)$ and $\mathsf{Ran}(\beta'^{\triangleright}) \subseteq \mathsf{Dom}(\eta_2)$,*
3. *$\mathsf{Dom}(\beta'^{\triangleleft}) \subseteq \mathsf{Dom}(T')$ and $\mathsf{Dom}(\beta'^{\triangleright}) \subseteq \mathsf{Dom}(T'')$*

Then $\beta', (T', T'') \vdash \eta_1 \simeq^{\lambda} \eta_2$

Proof. β' is well-defined w.r.t. η_1, η_2, T' and T'' follow directly from the β' definition. For the second item we proceed as follows. Let $o \in \mathsf{Dom}(\beta'_{loc})$ and let $f \in \mathsf{Dom}(\eta_1(o))$. We must prove:

$$\beta', \lfloor T'(\beta'^{\triangleleft -1}(o), f) \rfloor \vdash \eta_1(o, f) \simeq^{\lambda} \eta_2(\beta'(o), f).$$

This result follow by hypothesis $T \leq T'$ and $T \leq T''$, i.e. $\forall a \in \mathsf{Dom}(T).\forall f \in \mathbb{F}.\lfloor T(a, f) \rfloor = \lfloor T'(a, f) \rfloor = \lfloor T''(a, f) \rfloor$. ▪

We recall from Definition 3.27 that β^{id} is a location bijection bijection set such that $\beta^{\mathsf{id}}_{loc}$ is the identity over the domain of β_{loc}.

Lemma B.17. *If $\beta^{\mathsf{id}}, (T, T) \vdash \eta_1 \simeq^{\lambda} \eta_2$ and $T \leq T'$, then $\beta^{\mathsf{id}}, (T, T') \vdash \eta_1 \simeq^{\lambda} \eta_2$.*

Proof. Immediate from the fact that for any $a \in \mathsf{Dom}(T)$ and $f \in \mathsf{Dom}(T(a))$, $\lfloor T(a, f) \rfloor = \lfloor T'(a, f) \rfloor$ and hypothesis $T \leq T'$. ▪

The following lemma shows that transitions from some high region for k to a low one through $i \mapsto i'$, implies that i' is a junction point for the latter region.

Lemma B.18. *Let $M[\overrightarrow{\alpha}]$ be a well-typed method with code $B[\overrightarrow{\alpha}]$ and $i \in \mathsf{region}(k)$ for some $k \in \mathsf{Dom}(B)^\sharp$. Suppose $l \notin \mathtt{H}_I$ and $\forall k' \in \mathsf{region}(k).l \sqsubseteq \mathcal{A}_{k'}$ and furthermore let $i \mapsto i'$, $\mathcal{A}_i \notin \mathtt{H}_I$ and $\mathcal{A}_{i'} \in \mathtt{H}_I$. Then $i' = \mathsf{jun}(k)$.*

Proof. Suppose $i' \neq \mathsf{jun}(k)$. By SOAP (property 3.10(2)) $i' \in \mathsf{region}(k)$. Furthermore, by the hypothesis $l \notin \mathtt{H}_I$ and $\forall k' \in \mathsf{region}(k).l \sqsubseteq \mathcal{A}_{k'}$ we have $l \sqsubseteq \mathcal{A}_{i'}$. However, this contradicts $\mathcal{A}_{i'} \in \mathtt{H}_I$.

▪

B.3 Proof of Unwinding Lemmas

We now address the proofs of the Lemma 4.20, Lemma 4.21 and Lemma 4.22. These proofs are similar to the proofs of the unwinding lemmas presented in Appendix A.

Lemma 4.20 (One-Step Preservation of Equality of High Integrity Data on Low Context) Suppose

1. $\mathcal{A}_{s_1} \in \mathtt{H}_I$ and $\mathtt{pc}(s_1) = \mathtt{pc}(s_2)$
2. $s_1 \longrightarrow_{B[\overrightarrow{a_1}]} s_1'$;
3. $s_2 \longrightarrow_{B[\overrightarrow{a_2}]} s_2'$; and
4. $\beta \vdash s_1 \simeq^{\mathtt{A}} s_2$ for some location bijection set β.

Then
$\beta' \vdash s_1' \simeq^{\mathtt{A}} s_2'$ and $\mathtt{pc}(s_1') = \mathtt{pc}(s_2')$, for some $\beta' \supseteq \beta$.

Proof. It proceeds by case analysis on the instruction that is executed. We supply two sample cases.

Case: Suppose $B(i) = \mathtt{store}\ x$. Then

$$\frac{B(i) = \mathtt{store}\ x}{\langle i, \alpha_1, v_1 \cdot \sigma_1, \eta_1\rangle \longrightarrow_B \langle i+1, \alpha_1 \oplus \{x \mapsto v_1\}, \sigma_1, \eta_1\rangle = s_1'}$$

$$\frac{B(i) = \mathtt{store}\ x}{\langle i, \alpha_2, v_2 \cdot \sigma_2, \eta_2\rangle \longrightarrow_B \langle i+1, \alpha_2 \oplus \{x \mapsto v_2\}, \sigma_2, \eta_2\rangle = s_2'}$$

Moreover, by T-Store:

$$\begin{array}{l} \mathcal{S}_i \leq_{A_i} \mathcal{V}_{i+1}(x) \cdot \mathcal{S}_{i+1} \\ \mathcal{V}_i \setminus x \leq \mathcal{V}_{i+1} \setminus x \\ \mathcal{A}_i \sqsubseteq \mathcal{V}_{i+1}(x) \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \end{array}$$

We prove $\beta \vdash s_1' \simeq^\lambda s_2'$.

1. Since $i_1' = i + 1 = i_2'$, then $\mathcal{A}_{i_1'} = \mathcal{A}_{i_2'}$.
2. First we note that $\beta, \lfloor \mathcal{V}_{i+1}(x) \rfloor \vdash v_1 \simeq^\lambda v_2$ **(1)** follows from hypothesis $\beta, \lfloor \mathcal{S}_i(0) \rfloor \vdash v_1 \simeq^\lambda v_2$, condition 1 of T-Store ($\mathcal{S}_i(0) \sqsubseteq \mathcal{V}_{i+1}(x)$) and Lemma B.7 .
Furthermore, by hypothesis $\beta, \mathcal{V}_i \setminus x, \lfloor \mathcal{A}_i \rfloor \vdash \alpha_1 \setminus x \simeq^\lambda \alpha_2 \setminus x$, $\mathcal{A}_i \in \mathtt{H}_I$ and by condition 2 of T-Store and Lemma B.8 we have $\beta, \mathcal{V}_{i+1} \setminus x, \mathcal{A}_i \vdash \alpha_1 \setminus x \simeq^\lambda \alpha_2 \setminus x$ **(2)**.
From **(1)** and **(2)** we deduce $\beta, \mathcal{V}_{i+1}, \mathcal{A}_i \vdash \alpha_1 \simeq^\lambda \alpha_2$ **(3)**.
If $\mathcal{A}_{i+1} \in \mathtt{H}_I$ then $\beta, \mathcal{V}_{i+1}, \mathcal{A}_{i+1} \vdash \alpha_1 \simeq^\lambda \alpha_2$ follows by **(3)** and Lemma B.11.
If $\mathcal{A}_{i+1} \notin \mathtt{H}_I$, then we note that $\beta, \mathcal{V}_{i+1}, \mathcal{A}_{i+1} \vdash \alpha_1 \simeq^\lambda \alpha_2$ follows from **(3)**.

3. By hypothesis $\beta, \mathcal{S}_i, \mathcal{A}_i \vdash v_1 \cdot \sigma_1 \simeq^\lambda v_2 \cdot \sigma_2$, condition 1 of T-Store and lemma B.12 we have $\beta, \mathcal{V}_{i+1}(x) \cdot \mathcal{S}_{i+1}, \mathcal{A}_i \vdash v_1 \cdot \sigma_1 \simeq^\lambda v_2 \cdot \sigma_2$.
   Now, by $\mathcal{A}_i \in \mathtt{H}_I$ and Cons-L-I we have $\beta, \mathcal{S}_{i+1}, \mathcal{A}_i \vdash \sigma_1 \simeq^\lambda \sigma_2$.
   We have two cases:
   - $\mathcal{A}_{i+1} \in \mathtt{H}_I$. Then by Lem. B.15 we have $\beta, \mathcal{S}_{i+1}, \mathcal{A}_{i+1} \vdash \sigma_1 \simeq^\lambda \sigma_2$, as required.
   - $\mathcal{A}_{i+1} \notin \mathtt{H}_I$. Then by H-Low-I $\beta, (\mathcal{S}_{i+1}, \mathcal{S}_{i+1}), \mathcal{A}_{i+1} \vdash \sigma_1 \simeq^\lambda \sigma_2$ holds.
4. By hypothesis $\beta, \mathcal{T}_i \vdash \eta_1 \simeq^\lambda \eta_2$, condition 4 of `T-Store` and Lemma B.16, $\beta, \mathcal{T}_{i+1} \vdash \eta_1 \simeq^\lambda \eta_2$.

Case: $B(i) = \mathtt{load}\ x$. By the operational semantics:

$$\frac{B(i) = \mathtt{load}\ x}{\langle i, \alpha_1, \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i+1, \alpha_1, \alpha_1(x) \cdot \sigma_1, \eta_1 \rangle = s_1'}$$

$$\frac{B(i) = \mathtt{load}\ x}{\langle i, \alpha_2, \sigma_2, \eta_2 \rangle \longrightarrow_B \langle i+1, \alpha_2, \alpha_2(x) \cdot \sigma_2, \eta_2 \rangle = s_2'}$$

Moreover, by T-Load:

$$\begin{array}{l} \mathcal{A}_i \sqcup \mathcal{V}_i(x) \cdot \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ \mathcal{T}_i \leq \mathcal{T}_{i+1} \end{array}$$

We prove $\beta \vdash s_1' \simeq^\lambda s_2'$.

1. By the operational semantics $i_1' = i + 1 = i_2'$, hence $\mathcal{A}_{i_1'} = \mathcal{A}_{i_1'}$.
2. We must prove $\beta, \mathcal{V}_{i+1}, \lfloor \mathcal{A}_{i+1} \rfloor \vdash \alpha_1 \simeq^\lambda \alpha_2$. Given that $\mathcal{A}_{i+1}$ may be either low or high we must consider both cases. Therefore, suppose first that $\mathcal{A}_{i+1} \in \mathtt{H}_I$. By hypothesis we know $\beta, \mathcal{V}_i, \lfloor \mathcal{A}_i \rfloor \vdash \alpha_1 \simeq^\lambda \alpha_2$, $\mathcal{A}_i \in \mathtt{H}_I$. Then, resorting to condition 2 of T-Load and Lemma B.8, $\beta, \mathcal{V}_{i+1}, \lfloor \mathcal{A}_i \rfloor \vdash \alpha_1 \simeq^\lambda \alpha_2$ Now, by Lemma B.11 the result follows.
   Suppose now that $\mathcal{A}_{i+1} \notin \mathtt{H}_I$. We proceed as above and simply note that by the definition of indistinguishability of local variable frames $\beta, \mathcal{V}_{i+1}, \lfloor \mathcal{A}_{i+1} \rfloor \vdash \alpha_1 \simeq^\lambda \alpha_2$ follows from $\beta, \mathcal{V}_{i+1}, \lfloor \mathcal{A}_i \rfloor \vdash \alpha_1 \simeq^\lambda \alpha_2$.
3. We have two cases:
   - $\mathcal{A}_{i+1} \in \mathtt{H}_I$. By hypothesis we know $\beta, \mathcal{S}_i, \mathcal{A}_i \vdash \sigma_1 \simeq^\lambda \sigma_2$, $\mathcal{A}_i \in \mathtt{H}_I$ and $\beta, \lfloor \mathcal{V}_i(x) \rfloor \vdash \alpha_1(x) \simeq^\lambda \alpha_2(x)$. From the latter and Lemma B.7 we deduce $\beta, \lfloor \mathcal{A}_i \sqcup \mathcal{V}_i(x) \rfloor \vdash \alpha_1(x) \simeq^\lambda \alpha_2(x)$ Then, by L-Cons-I,
   $$\beta, \mathcal{A}_i \sqcup \mathcal{V}_i(x) \cdot \mathcal{S}_i, \mathcal{A}_i \vdash \alpha_1(x) \cdot \sigma_1 \simeq^\lambda \alpha_2(x) \cdot \sigma_2 \tag{B.1}$$
   Condition 1 of T-Load, namely $\mathcal{A}_i \sqcup \mathcal{V}_i(x) \cdot \mathcal{S}_i \leq_{\mathcal{A}_i} \mathcal{S}_{i+1}$, and (B.1) allows us to apply Lemma B.12 to deduce $\beta, \mathcal{S}_{i+1}, \mathcal{A}_i \vdash \alpha_1(x) \cdot \sigma_1 \simeq^\lambda \alpha_2(x) \cdot \sigma_2$. Now, by Lem. B.15 follows the result.
   - $\mathcal{A}_{i+1} \notin \mathtt{H}_I$. By the previous case we know $\beta, \mathcal{S}_{i+1}, \mathcal{A}_i \vdash \alpha_1(x) \cdot \sigma_1 \simeq^\lambda \alpha_2(x) \cdot \sigma_2$. Then by H-Low-I $\beta, (\mathcal{S}_{i+1}, \mathcal{S}_{i+1}), \mathcal{A}_{i+1} \vdash \alpha_1(x) \cdot \sigma_1 \simeq^\lambda \alpha_2(x) \cdot \sigma_2$ holds.
4. By hypothesis $\beta, \mathcal{T}_i \vdash \eta_1 \simeq^\lambda \eta_2$, condition 3 of `T-Load` and Lemma B.16, $\beta, \mathcal{T}_{i+1} \vdash \eta_1 \simeq^\lambda \eta_2$.

∎

Lemma 4.21 (One-Step Preservation of Equality of High Integrity Data on High Context)

Let $\mathtt{pc}(s_1), \mathtt{pc}(s_1'), \mathtt{pc}(s_2) \in \mathtt{L}_I$. Furthermore, suppose:

1. $\beta \vdash s_1 \simeq^{\mathtt{A}} s_1'$ for some location bijection set β;
2. $s_1 \longrightarrow_{B[\overrightarrow{a}]} s_2$.

Then $\beta \vdash s_2 \simeq^{\mathtt{A}} s_1'$.

Proof. First we prove $\beta^{\mathsf{id}} \vdash s_1 \simeq^{\lambda} s_1'$ by a case analysis on the instruction that is executed and we conclude $\beta \circ \beta^{\mathsf{id}} \vdash s_1' \simeq^{\lambda} s_2$ by the typing rule and the third item of the Lemma B.6 (transitivity). We note that by β^{id} definition $\beta \circ \beta^{\mathsf{id}} = \beta$ then we conclude $\beta \vdash s_1' \simeq^{\lambda} s_2$

Before commencing, we point out that $\mathcal{A}_{i_1}, \mathcal{A}_{i_2}, \mathcal{A}_{i_1'} \notin \mathtt{H}_I$. In particular, $\mathcal{A}_{i_1}, \mathcal{A}_{i_1'} \notin \mathtt{H}_I$ and thus this proves the first item needed to determined that s_1 and s_1' are indistinguishable. We consider the remaining three for each reduction steps case.

Case: $B(i_1) = \mathtt{store}\ x$

$$\frac{B(i_1) = \mathtt{store}\ x}{\langle i_1, \alpha_1, v_1 \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i_1 + 1, \alpha_1 \oplus \{x \mapsto v_1\}, \sigma_1, \eta_1 \rangle = s_1'}$$

Moreover, by T-Store:

$$\begin{array}{l} \mathcal{S}_{i_1} \leq_{\mathcal{A}_{i_1}} \mathcal{V}_{i_1+1}(x) \cdot \mathcal{S}_{i_1+1} \\ \mathcal{V}_{i_1} \setminus x \leq \mathcal{V}_{i_1+1} \setminus x \\ \mathcal{A}_{i_1} \sqsubseteq \mathcal{V}_{i_1+1}(x) \\ \mathcal{T}_{i_1} \leq \mathcal{T}_{i_1+1} \end{array}$$

We prove $\beta^{\mathsf{id}} \vdash s_1 \simeq^{\lambda} s_1'$ by considering the two remaining items of machine state indistinguishability:

1. First we $\beta^{\mathsf{id}}, (\mathcal{V}_{i_1}, \mathcal{V}_{i_1+1}), \mathcal{A}_{i_1+1} \vdash \alpha_1 \simeq^{\lambda} \alpha_1 \oplus \{x \mapsto v_1\}$.
The only variable of interest is x since it is the only one that is changed. From condition 3 of T-Store, $\mathcal{A}_{i_1} \sqsubseteq \mathcal{V}_{i_1+1}(x)$ and hence $\beta^{\mathsf{id}}, \lfloor (\mathcal{V}_{i_1}(x) \sqcup \mathcal{V}_{i_1+1}(x)) \rfloor \vdash \alpha_1(x) \simeq^{\lambda} \alpha_1 \oplus \{x \mapsto v_1\}(x)$ **(1)**.
Now, by Lemma B.3 (reflexivity of variable high indistinguishability)
$$\beta^{\mathsf{id}}, (\mathcal{V}_{i_1} \setminus x, \mathcal{V}_{i_1} \setminus x), \mathcal{A}_{i_1} \vdash \alpha_1 \setminus x \simeq^{\lambda} \alpha_1 \oplus \{x \mapsto v_1\} \setminus x$$
By this and by condition 2 of T-Store and by Lemma B.9 we have $\beta^{\mathsf{id}}, (\mathcal{V}_{i_1} \setminus x, \mathcal{V}_{i_1+1} \setminus x), \mathcal{A}_{i_1} \vdash \alpha_1 \setminus x \simeq^{\lambda} \alpha_1 \oplus \{x \mapsto v_1\} \setminus x$ **(2)**. By **(2)**, **(1)** and indistinguishability definition, we conclude that $\beta^{\mathsf{id}}, (\mathcal{V}_{i_1}, \mathcal{V}_{i_1+1}), \mathcal{A}_{i_1+1} \vdash \alpha_1 \simeq^{\lambda} \alpha_1 \oplus \{x \mapsto v_1\}$.
2. By Lemma B.4 (reflexivity of stack high indistinguishability) $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1} \setminus 0, \mathcal{S}_{i_1} \setminus 0), \mathcal{A}_{i_1} \vdash \sigma_1 \simeq^{\lambda} \sigma_1$.
By condition 1 of T-Store, $\mathcal{S}_{i_1} \setminus 0 \leq_{\mathcal{A}_{i_1}} \mathcal{S}_{i_1+1}$. Therefore, by resorting to Lemma B.13 we deduce
$$\beta^{\mathsf{id}}, (\mathcal{S}_{i_1} \setminus 0, \mathcal{S}_{i_1+1}), \mathcal{A}_{i_1} \vdash \sigma_1 \simeq^{\lambda} \sigma_1$$
Finally, by H-Cons-L-I
$$\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1+1}), \mathcal{A}_{i_1+1} \vdash v_1 \cdot \sigma_1 \simeq^{\lambda} \sigma_1$$
3. By definition of β^{id} and Lemma B.5 (reflexivity of heap high indistinguishability) we have that $\beta^{\mathsf{id}}, (\mathcal{T}_{i_1}, \mathcal{T}_{i_1}) \vdash \eta_1 \simeq^{\lambda} \eta_1$. Now, by the hypothesis $\mathcal{T}_{i_1} \leq \mathcal{T}_{i_1+1}$ and the Lemma B.17 we deduce $\beta^{\mathsf{id}}, (\mathcal{T}_{i_1}, \mathcal{T}_{i_1+1}) \vdash \eta_1 \simeq^{\lambda} \eta_1$.

Case: $B(i_1) = \mathtt{load}\ x$. By the operational semantics :

$$\frac{B(i_1) = \mathtt{load}\ x}{\langle i_1, \alpha_1, \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i_1 + 1, \alpha_1, \alpha_1(x) \cdot \sigma_1, \eta_1 \rangle = s_1'}$$

Moreover, from T-Load:

$$\begin{array}{l} \mathcal{A}_{i_1} \sqcup \mathcal{V}_{i_1}(x) \cdot \mathcal{S}_{i_1} \leq_{\mathcal{A}_{i_1}} \mathcal{S}_{i_1+1} \\ \mathcal{V}_{i_1} \leq \mathcal{V}_{i_1+1} \\ \mathcal{T}_{i_1} \leq \mathcal{T}_{i_1+1} \end{array}$$

We prove the three remaining items of $\beta^{\mathsf{id}} \vdash s_1 \simeq^{\lambda} s_1'$.

1. We address $\beta^{\mathsf{id}}, (\mathcal{V}_{i_1}, \mathcal{V}_{i_1+1}), \mathcal{A}_{i_1+1} \vdash \alpha_1 \simeq^{\lambda} \alpha_1$.
   By Lemma B.3 (reflexivity of variable high indistinguishability), $\beta^{\mathsf{id}}, (\mathcal{V}_{i_1}, \mathcal{V}_{i_1}), \mathcal{A}_{i_1} \vdash \alpha_1 \simeq^{\lambda} \alpha_1$. By this and by condition 2 of T-Load and by Lemma B.9, $\beta^{\mathsf{id}}, (\mathcal{V}_{i_1}, \mathcal{V}_{i_1+1}), \mathcal{A}_{i_1} \vdash \alpha_1 \simeq^{\lambda} \alpha_1$.
2. By Lemma B.4 (reflexivity of stack high indistinguishability) we have that $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1}), \mathcal{A}_{i_1} \vdash \sigma_1 \simeq^{\lambda} \sigma_1$. Given that $\mathcal{A}_{i_1} \notin \mathtt{H}_I$ we may resort to H-Cons-R-I to obtain $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{A}(i) \sqcup \mathcal{V}_{i_1}(x) \cdot \mathcal{S}_{i_1})), \mathcal{A}_{i_1} \vdash \sigma_1 \simeq^{\lambda} \alpha_1(x) \cdot \sigma_1$. Finally, by condition 1 of T-Load and Lemma B.13 we have $\beta^{\mathsf{id}}, (\mathcal{S}_{i_1}, \mathcal{S}_{i_1+1}), \mathcal{A}(i_1+1) \vdash \sigma_1 \simeq^{\lambda} \alpha_1(x) \cdot \sigma_1$.
3. Idem to `store` case.

▪

Lemma 4.22 (One-Step Preservation of Equality of High Integrity Data on High to Low Context)

Let $\mathtt{pc}(s_1), \mathtt{pc}(s_2) \in \mathtt{L}_I$ and $\mathtt{pc}(s_1'), \mathtt{pc}(s_2') \in \mathtt{H}_I$. Furthermore, suppose:

1. $s_1 \longrightarrow_{B[\overrightarrow{a_1}]} s_1'$:
2. $s_2 \longrightarrow_{B[\overrightarrow{a_2}]} s_2'$;
3. $\beta \vdash s_1 \simeq^{\mathtt{A}} s_2$, for some location bijection set β; and
4. $\mathtt{pc}(s_1') = \mathtt{pc}(s_2')$.

Then $\beta \vdash s_1' \simeq^{\mathtt{A}} s_2'$.

Proof. There are several options to consider.

- Due to our assumption on the forms of programs (see comment below Property 3.10), we may assume both $B(i_1) = \mathtt{goto}\ i_1'$ and $B(i_2) = \mathtt{goto}\ i_2'$. Thus, $s_1' = \langle i_1', \alpha_1, \sigma_1, \eta_1 \rangle$ and $s_2' = \langle i_2', \alpha_2, \sigma_2, \eta_2 \rangle$ and both of the following hold:

$$\frac{B(i_1) = \mathtt{goto}\ i_1'}{\langle i_1, \alpha_1, \sigma_1, \eta_1 \rangle \longrightarrow_B \langle i_1', \alpha_1, \sigma_1, \eta_1 \rangle} \qquad \frac{B(i_2) = \mathtt{goto}\ i_2'}{\langle i_2, \alpha_2, \sigma_2, \eta_2 \rangle \longrightarrow_B \langle i_2', \alpha_2, \sigma_2, \eta_2 \rangle}$$

  Also, from T-Goto:

$$\begin{array}{ll} \mathcal{V}_{i_1} \leq \mathcal{V}_{i_1'} & \mathcal{V}_{i_2} \leq \mathcal{V}_{i_2'} \\ \mathcal{S}_{i_1} \leq_{\mathcal{A}_{i_1}} \mathcal{S}_{i_1'} & \mathcal{S}_{i_2} \leq_{\mathcal{A}_{i_2}} \mathcal{S}_{i_2'} \\ \mathcal{T}_{i_1} \leq \mathcal{T}_{i_1'} & \mathcal{T}_{i_2} \leq \mathcal{T}_{i_2'} \end{array}$$

  We now address the proof of both items:

  I- $i_1' = \mathsf{jun}(k) = i_2'$. By the hypothesis $\mathcal{A}_{i_1}, \mathcal{A}_{i_2} \notin \mathtt{H}_I$, $i_1 \mapsto i_1'$, $i_2 \mapsto i_2'$ and by assumption $\mathcal{A}_{i_1'} = \mathcal{A}(i_2') \in \mathtt{H}_I$. Therefore, by Lemma B.18 we know that $i_1' = \mathsf{jun}(k)$ and that $i_2' = \mathsf{jun}(k)$. But by the SOAP property, junction points are unique. Hence $i_1' = \mathsf{jun}(k) = i_2'$.

  II- We address the proof of the four items of the definition of state indistinguishability for $\beta \vdash s_1' \simeq^{\lambda} s_2'$:

  1. By hypothesis $\mathcal{A}_{i_1'} = \mathcal{A}_{i_2'}$ and $\mathcal{A}_{i_1'} \in \mathtt{H}_I$.
  2. We must prove $\beta, \mathcal{V}_{i_1'}, \mathcal{A}_{i_1'} \vdash \alpha_1 \simeq^{\lambda} \alpha_2$ (since $\alpha_1' = \alpha_1$ and $\alpha_2' = \alpha_2$). Recall from above that by hypothesis:
  $$\beta, (\mathcal{V}_{i_1}, \mathcal{V}_{i_2}), l \vdash \alpha_1 \simeq^{\lambda} \alpha_2, l \notin \mathtt{H}_I \qquad \text{(B.2)}$$
     Let $x \in \mathbb{X}$. We have two cases:
     - If $\mathcal{V}_{i_1'}(x) \notin \mathtt{H}_I$, then we trivially have $\beta, \lfloor \mathcal{V}_{i_1'}(x) \rfloor \vdash \alpha_1(x) \simeq^{\lambda} \alpha_2(x)$.
     - If $\mathcal{V}_{i_1'}(x) \in \mathtt{H}_I$, then $\mathcal{V}_{i_1}(x) \in \mathtt{H}_I$ and $\mathcal{V}_{i_2}(x) \in \mathtt{H}_I$. Thus $\beta, \lfloor \mathcal{V}_{i_1'}(x) \rfloor \vdash \alpha_1(x) \simeq^{\lambda} \alpha_2(x)$ follows from (B.2).

3. We must prove $\beta, \mathcal{S}_{i'_1}, \mathcal{A}_{i'_1} \vdash \sigma_1 \simeq^{\lambda} \sigma_2$. By the hypothesis,

$$\beta, (\mathcal{S}_{i_1}, \mathcal{S}_{i_2}), l \vdash \sigma_1 \simeq^{\lambda} \sigma_2, l \notin \mathtt{H}_I \tag{B.3}$$

and by condition 2 of T-Goto we know that $\|\mathcal{S}_{i_1}\| = \|\mathcal{S}_{i'_1}\| = \|\mathcal{S}_{i_2}\|$ (also, $\mathcal{S}_{i'_1} = \mathcal{S}_{i'_2}$).
For each $j \in \{0..\|\mathcal{S}_{i'_1}\| - 1\}$, we proceed as follows:
   - If $\mathcal{S}_{i'_1}(j) \notin \mathtt{H}_I$, then trivially $\beta, \lfloor \mathcal{S}_{i'_1}(j) \rfloor \vdash \sigma_1(j) \simeq^{\lambda} \sigma_2(j)$.
   - If $\mathcal{S}_{i'_1}(j) \in \mathtt{H}_I$, then $\mathcal{S}_{i_1}(j) = \mathcal{S}_{i_2}(j) \in \mathtt{H}_I$. From (B.3) and the definition of stack indistinguishability, it follows that there exists $k \geq j$ such that the prefix of size k of σ_1 and σ_2 are low indistinguishable. As a consequence, $\beta, \lfloor \mathcal{S}_{i'_1}(j) \rfloor \vdash \sigma_1(j) \simeq^{\lambda} \sigma_2(j)$.

   Hence, $\beta, \mathcal{S}_{i'_1}, \mathcal{A}_{i'_1} \vdash \sigma_1 \simeq^{\lambda} \sigma_2$.
4. We address the two items of the definition of $\beta, \mathcal{T}_{i'_1} \vdash \eta_1 \simeq^{\lambda} \eta_2$. The first follow from the hypothesis that s_1 and s_2 are indistinguishable and $\mathcal{T}_{i_1}, \mathcal{T}_{i_2} \leq \mathcal{T}_{i'_1}$. Regarding the last item, we proceed as follows. Let $o \in \mathsf{Dom}(\beta_{loc})$, and let $f \in \mathsf{Dom}(\eta_1(o))$. We must prove that:
$$\beta, \lfloor \mathcal{T}_{i'_1}(\beta^{\lhd -1}(o), f) \rfloor \vdash \eta_1(o, f) \simeq^{\lambda} \eta_2(\beta(o), f).$$
This is an immediate consequence of the hypothesis, $\sim^{\lambda}_{\beta} \mathcal{T}_{i_1}(\beta^{\lhd -1}(o), f)\eta_1(o, f)\eta_2(\beta(o), f)$ and $\mathcal{T}_{i_1}, \mathcal{T}_{i_2} \leq \mathcal{T}_{i'_1}$.

- We may assume both $B(i_1) = \mathtt{return}$ and $B(i_2) = \mathtt{return}$. Thus, $s'_1 = \langle p_f, v_1, \eta_1 \rangle$ and $s'_2 = \langle p_f, v_2, \eta_2 \rangle$ and both of the following hold:

$$\frac{B(i_1) = \mathtt{return}}{\langle i_1, \alpha_1, v_1 \cdot \sigma_1, \eta_1 \rangle \longrightarrow_B \langle p_f, v_1, \eta_1 \rangle} \qquad \frac{B(i_2) = \mathtt{return}}{\langle i_2, \alpha_2, v_2 \cdot \sigma_2, \eta_2 \rangle \longrightarrow_B \langle p_f, v_2, \eta_2 \rangle}$$

  Also, from T-Ret:

$$\begin{array}{ll} \mathcal{S}_{i_1}(0) \sqcup \mathcal{A}_{i_1} \sqsubseteq \kappa_r & \mathcal{S}_{i_2}(0) \sqcup \mathcal{A}_{i_2} \sqsubseteq \kappa_r \\ \mathcal{T}_{i_1} \leq \mathcal{T}_{p_f} & \mathcal{T}_{i_2} \leq \mathcal{T}_{p_f} \end{array}$$

  We now address the proof of both items:

  I- By successor relation definition $\mathtt{pc}(s'_1)$=$\mathtt{pc}(s'_2)$=p_f.

  II- We address the proof of the two items of the definition of final state indistinguishability for $\beta \vdash s'_1 \simeq^{\lambda} s'_2$:
  1. By hypothesis $\mathcal{A}(p_f) \in \mathtt{H}_I$.
  2. We must prove $\beta, \lfloor \kappa_r \rfloor \vdash v_1 \simeq^{\lambda} v_2$. By the hypothesis, $\beta, (\mathcal{S}_{i_1}, \mathcal{S}_{i_2}), l \vdash v_1 \cdot \sigma_1 \simeq^{\lambda} v_2 \cdot \sigma_2$, $l \notin \mathtt{H}_I$, then we have $\beta, \lfloor \mathcal{S}_{i_1}(0) \sqcup \mathcal{S}_{i_2}(0) \rfloor \vdash v_1 \simeq^{\lambda} v_2$.
  Furthermore, by T-Ret $\mathcal{S}_{i_1}(0) \sqsubseteq \kappa_r$ and $\mathcal{S}_{i_2}(0) \sqsubseteq \kappa_r$.
  Now, by indistinguishability definition $\beta, \lfloor \kappa_r \rfloor \vdash v_1 \simeq^{\lambda} v_2$.
  3. Idem to previous `T-Goto` case.
- The remain cases $B(i_1) = \mathtt{return}$ and $\mathtt{pc}(i_2) = err_i$, $\mathtt{pc}(i_1) = err_i$, $B(i_2) = \mathtt{return}$ and $\mathtt{pc}(i_1) = err_i$, $\mathtt{pc}(i_2) = err_i$ are similar to the previous case.

▪

C

Definitions and Proofs for Justification Logic and Audited Computation

C.1 Basic Definitions

Definition C.1.1 (Free truth variables) The set of free truth variables in evidence s (likewise for equivalence witness e and term M) is denoted $\mathsf{fvT}(\mathrm{s})$ and defined as follows:

$$
\begin{aligned}
\mathsf{fvT}(a) &\stackrel{\text{def}}{=} \{a\} \\
\mathsf{fvT}(\lambda b : A.t) &\stackrel{\text{def}}{=} \mathsf{fvT}(t) \setminus \{b\} \\
\mathsf{fvT}(t_1 \cdot t_2) &\stackrel{\text{def}}{=} \mathsf{fvT}(t_1) \cup \mathsf{fvT}(t_2) \\
\mathsf{fvT}(\Sigma.t) &\stackrel{\text{def}}{=} \emptyset \\
\mathsf{fvT}(\mathrm{LET}(u^{A[\Sigma]}.t_2, t_1)) &\stackrel{\text{def}}{=} \mathsf{fvT}(t_1) \cup \mathsf{fvT}(t_2) \\
\mathsf{fvT}(\alpha\theta^w) &\stackrel{\text{def}}{=} \bigcup \mathsf{fvT}(\theta^w)
\end{aligned}
$$

For equivalence witnesses:

$$
\begin{aligned}
\mathsf{fvT}(\mathfrak{r}(t)) &\stackrel{\text{def}}{=} \mathsf{fvT}(t) \\
\mathsf{fvT}(\mathfrak{s}(e)) &\stackrel{\text{def}}{=} \mathsf{fvT}(e) \\
\mathsf{fvT}(\mathfrak{t}(e_1, e_2)) &\stackrel{\text{def}}{=} \mathsf{fvT}(e_1) \cup \mathsf{fvT}(e_2) \\
\mathsf{fvT}(\mathfrak{ba}(a^A.r, t)) &\stackrel{\text{def}}{=} \mathsf{fvT}(r) \setminus \{a\} \cup \mathsf{fvT}(t) \\
\mathsf{fvT}(\mathfrak{bb}(u^{A[\Sigma_1]}.r, \Sigma.t)) &\stackrel{\text{def}}{=} \mathsf{fvT}(r) \\
\mathsf{fvT}(\mathfrak{ti}(\theta^w, \alpha)) &\stackrel{\text{def}}{=} \bigcup_{i \in 1..10} \mathsf{fvT}(\theta^w(c_i)) \\
\mathsf{fvT}(\mathfrak{abC}(b^A.e)) &\stackrel{\text{def}}{=} \mathsf{fvT}(e) \setminus \{b\} \\
\mathsf{fvT}(\mathfrak{apC}(e_1, e_2)) &\stackrel{\text{def}}{=} \mathsf{fvT}(e_1) \cup \mathsf{fvT}(e_2) \\
\mathsf{fvT}(\mathfrak{leC}(u^{A[\Sigma]}.e_1, e_2)) &\stackrel{\text{def}}{=} \mathsf{fvT}(e_1) \cup \mathsf{fvT}(e_2) \\
\mathsf{fvT}(\mathfrak{rpC}(\overline{e})) &\stackrel{\text{def}}{=} \bigcup_{i \in 1..10} \mathsf{fvT}(e_i)
\end{aligned}
$$

For terms:

$$\begin{aligned}
\mathsf{fvT}(a) &\stackrel{\text{def}}{=} \{a\} \\
\mathsf{fvT}(\lambda a : A.M) &\stackrel{\text{def}}{=} \mathsf{fvT}(M) \setminus \{a\} \\
\mathsf{fvT}(M\,N) &\stackrel{\text{def}}{=} \mathsf{fvT}(M) \cup \mathsf{fvT}(N) \\
\mathsf{fvT}(\langle u; \sigma\rangle) &\stackrel{\text{def}}{=} \emptyset \\
\mathsf{fvT}(!^{\Sigma}_{e} M) &\stackrel{\text{def}}{=} \emptyset \\
\mathsf{fvT}(\mathsf{let}\, u : A[\Sigma] = M \;\mathsf{in}\; N) &\stackrel{\text{def}}{=} \mathsf{fvT}(M) \cup \mathsf{fvT}(N) \\
\mathsf{fvT}(\alpha\theta) &\stackrel{\text{def}}{=} \textstyle\bigcup_{i \in 1..10} \mathsf{fvT}(\theta(c_i)) \\
\mathsf{fvT}(e \rhd M) &\stackrel{\text{def}}{=} \mathsf{fvT}(e) \cup \mathsf{fvT}(M)
\end{aligned}$$

The set of free validity and trail variables are defined similarly.

Definition C.1.2 (Truth variable substitution II) The result of substituting all free occurrences of a in M by N, s is denoted $M^a_{N,s}$ and defined as follows:

$$\begin{aligned}
a^a_{N,s} &\stackrel{\text{def}}{=} N \\
b^a_{N,s} &\stackrel{\text{def}}{=} b \\
(\lambda b : A.M)^a_{N,s} &\stackrel{\text{def}}{=} \lambda b : A.M^a_{N,s} \\
(M_1\, M_2)^a_{N,s} &\stackrel{\text{def}}{=} {M_1}^a_{N,s}\, {M_2}^a_{N,s} \\
\langle u; \sigma\rangle^a_{N,s} &\stackrel{\text{def}}{=} \langle u; \sigma\rangle \\
(!^{\Sigma}_{e} M)^a_{N,s} &\stackrel{\text{def}}{=} !^{\Sigma}_{e} M \\
(\mathsf{let}\, u : A[\Sigma] = M_1 \;\mathsf{in}\; M_2)^a_{N,s} &\stackrel{\text{def}}{=} \mathsf{let}\, u : A[\Sigma] = {M_1}^a_{N,s} \;\mathsf{in}\; {M_2}^a_{N,s} \\
(\alpha\theta)^a_{N,s} &\stackrel{\text{def}}{=} \alpha\theta \\
(e \rhd M)^a_{N,s} &\stackrel{\text{def}}{=} e^a_s \rhd M^a_{N,s}
\end{aligned}$$

Definition C.1.3 (Trail Variable Renaming) Let σ be a renaming. We define $s\sigma$, $e\sigma$ and $M\sigma$ as follows:

$$\begin{aligned}
a\sigma &\stackrel{\text{def}}{=} a \\
(\lambda b : A.t)\sigma &\stackrel{\text{def}}{=} \lambda b : A.t\sigma \\
(t_1 \cdot t_2)\sigma &\stackrel{\text{def}}{=} t_1\sigma \cdot t_2\sigma \\
\langle u; \sigma'\rangle\sigma &\stackrel{\text{def}}{=} \langle u; \sigma \circ \sigma'\rangle \\
(\Sigma.t)\sigma &\stackrel{\text{def}}{=} \Sigma.t \\
(\textsc{let}(u^{A[\Sigma]}.t_2, t_1))\sigma &\stackrel{\text{def}}{=} \textsc{let}(u^{A[\Sigma]}.t_2\sigma, t_1\sigma) \\
(\alpha\theta^w)\sigma &\stackrel{\text{def}}{=} \sigma(\alpha)\theta^w
\end{aligned}$$

For equivalence witnesses:

$$
\begin{aligned}
\mathfrak{r}(t)\sigma &\stackrel{\text{def}}{=} \mathfrak{r}(t\sigma)\\
\mathfrak{s}(e)\sigma &\stackrel{\text{def}}{=} \mathfrak{s}(e\sigma)\\
\mathfrak{t}(e_1,e_2)\sigma &\stackrel{\text{def}}{=} \mathfrak{t}(e_1\sigma,e_2\sigma)\\
\mathfrak{ba}(a^A.t_1,t_2)\sigma &\stackrel{\text{def}}{=} \mathfrak{ba}(a^A.t_1\sigma,t_2\sigma)\\
\mathfrak{bb}(u^{A[\Sigma_1]}.s,\Sigma_1.t)\sigma &\stackrel{\text{def}}{=} \mathfrak{bb}(u^{A[\Sigma_1]}.s\sigma,t)\\
(\mathfrak{ti}(\theta^w,\alpha))\sigma &\stackrel{\text{def}}{=} \mathfrak{ti}(\theta^w,\sigma(\alpha))\\
\mathfrak{ab}\mathfrak{C}(b^A.e)\sigma &\stackrel{\text{def}}{=} \mathfrak{ab}\mathfrak{C}(b^A.e\sigma)\\
\mathfrak{ap}\mathfrak{C}(e_1,e_2)\sigma &\stackrel{\text{def}}{=} \mathfrak{ap}\mathfrak{C}(e_1\sigma,e_2\sigma)\\
\mathfrak{le}\mathfrak{C}(u^{A[\Sigma]}.e_1,e_2)\sigma &\stackrel{\text{def}}{=} \mathfrak{le}\mathfrak{C}(u^{A[\Sigma]}.e_1\sigma,e_2\sigma)\\
\mathfrak{rp}\mathfrak{C}(\overline{e})\sigma &\stackrel{\text{def}}{=} \mathfrak{rp}\mathfrak{C}(\overline{e}\sigma)
\end{aligned}
$$

For terms:

$$
\begin{aligned}
a\sigma &\stackrel{\text{def}}{=} a\\
(\lambda a:A.M)\sigma &\stackrel{\text{def}}{=} \lambda a:A.M\sigma\\
(M_1\,M_2)\sigma &\stackrel{\text{def}}{=} (M_1\sigma\,M_2\sigma)\\
\langle u;\sigma'\rangle\sigma &\stackrel{\text{def}}{=} \langle u;\sigma\circ\sigma'\rangle\\
(!^{\Sigma}_{e}M)\sigma &\stackrel{\text{def}}{=} !^{\Sigma}_{e}M\\
(\mathsf{let}\,u:A[\Sigma]=M_1\;\mathsf{in}\;M_2)\sigma &\stackrel{\text{def}}{=} \mathsf{let}\,u:A[\Sigma]=M_1\sigma\;\mathsf{in}\;M_2\sigma\\
(\alpha\theta)\sigma &\stackrel{\text{def}}{=} \sigma(\alpha)\theta\\
(e\rhd M)\sigma &\stackrel{\text{def}}{=} e\sigma\rhd M\sigma
\end{aligned}
$$

Definition C.1.4 (Valid variable substitution I) The result of substituting all free occurrences of u in proposition P (resp. evidence t or equivalence witness e) by $\Sigma.s$ is denoted $P^u_{\Sigma.s}$ (resp. $t^u_{\Sigma.s}$ and $e^u_{\Sigma.s}$) and defined as follows:

$$
\begin{aligned}
P^u_{\Sigma.s} &\stackrel{\text{def}}{=} P\\
(A\supset B)^u_{\Sigma.s} &\stackrel{\text{def}}{=} A^u_{\Sigma.s}\supset B^u_{\Sigma.s}\\
([\![\Sigma'.t]\!]A)^u_{\Sigma.s} &\stackrel{\text{def}}{=} [\![\Sigma'.t^u_{\Sigma.s}]\!]A^u_{\Sigma.s}
\end{aligned}
$$

For equivalence witnesses:

$$
\begin{aligned}
\mathfrak{r}(t)^u_{\Sigma.s} &\stackrel{\text{def}}{=} \mathfrak{r}(t^u_{\Sigma.s})\\
\mathfrak{s}(e)^u_{s} &\stackrel{\text{def}}{=} \mathfrak{s}(e^u_{\Sigma.s})\\
\mathfrak{t}(e_1,e_2)^u_{\Sigma.s} &\stackrel{\text{def}}{=} \mathfrak{t}({e_1}^u_{\Sigma.s},{e_2}^u_{\Sigma.s})\\
\mathfrak{ba}(a^A.r,t)^u_{\Sigma.s} &\stackrel{\text{def}}{=} \mathfrak{ba}(a^A.r^u_{\Sigma.s},t^u_{\Sigma.s})\\
\mathfrak{bb}(v^{A[\Sigma']}.r,\Sigma'.t)^u_{\Sigma.s} &\stackrel{\text{def}}{=} \mathfrak{bb}(v^{A[\Sigma']}.r^u_{\Sigma.s},(\Sigma'.t)^u_{\Sigma.s})\\
\mathfrak{ti}(\theta^w,\alpha)^u_{\Sigma.s} &\stackrel{\text{def}}{=} \mathfrak{ti}({\theta^w}^u_{\Sigma.s},\alpha)\\
\mathfrak{ab}\mathfrak{C}(b^A.e)^u_{\Sigma.s} &\stackrel{\text{def}}{=} \mathfrak{ab}\mathfrak{C}(b^A.e^u_{\Sigma.s})\\
\mathfrak{ap}\mathfrak{C}(e_1,e_2)^u_{\Sigma.s} &\stackrel{\text{def}}{=} \mathfrak{ap}\mathfrak{C}({e_1}^u_{\Sigma.s},{e_2}^u_{\Sigma.s})\\
\mathfrak{le}\mathfrak{C}(v^{A[\Sigma']}.e_1,e_2)^u_{\Sigma.s} &\stackrel{\text{def}}{=} \mathfrak{le}\mathfrak{C}(v^{A[\Sigma']}.{e_1}^u_{\Sigma.s},{e_2}^a_{\Sigma.s})\\
\mathfrak{rp}\mathfrak{C}(\overline{e})^u_{\Sigma.s} &\stackrel{\text{def}}{=} \mathfrak{rp}\mathfrak{C}(\overline{e}^u_{\Sigma.s})
\end{aligned}
$$

Definition C.1.5 (Valid variable substitution II) The result of substituting all free occurrences of u in term M by $\Sigma.(N,t)$ is denoted $M^u_{\Sigma.(N,t,e)}$ and defined as follows:

$$\begin{aligned}
b^{u}_{\Sigma.(N,t,e)} &\stackrel{\text{def}}{=} b\\
(\lambda b : A.M)^{u}_{\Sigma.(N,t,e)} &\stackrel{\text{def}}{=} \lambda b : A.M^{u}_{\Sigma.(N,t,e)}\\
(M_1\, M_2)^{u}_{\Sigma.(N,t,e)} &\stackrel{\text{def}}{=} {M_1}^{u}_{\Sigma.(N,t,e)}\, {M_2}^{u}_{\Sigma.(N,t,e)}\\
\langle u;\sigma\rangle^{u}_{\Sigma.(N,t,e)} &\stackrel{\text{def}}{=} e\sigma \rhd N\sigma\\
\langle v;\sigma\rangle^{u}_{\Sigma.(N,t,e)} &\stackrel{\text{def}}{=} \langle v;\sigma\rangle\\
(!^{\Sigma'}_{e'} M)^{u}_{\Sigma.(N,t,e)} &\stackrel{\text{def}}{=} !^{\Sigma'}_{{e'}^{u}_{\Sigma.t}} M^{u}_{\Sigma.(N,t,e)}\\
(\mathsf{let}\, v : A[\Sigma'] = M_1 \,\mathsf{in}\, M_2)^{u}_{\Sigma.(N,t,e)} &\stackrel{\text{def}}{=} \mathsf{let}\, v : A[\Sigma'] = {M_1}^{u}_{\Sigma.(N,t,e)} \,\mathsf{in}\, {M_2}^{u}_{\Sigma.(N,t,e)}\\
(\alpha\theta)^{u}_{\Sigma.(N,t,e)} &\stackrel{\text{def}}{=} \alpha(\theta^{u}_{\Sigma.(N,t,e)})\\
(e' \rhd M)^{u}_{\Sigma.(N,t,e)} &\stackrel{\text{def}}{=} {e'}^{u}_{\Sigma.t} \rhd M^{u}_{\Sigma.(N,t,e)}
\end{aligned}$$

Definition C.1.6 (Trail constructor replacement) The result of replacing each equivalence witness constructor in e with a constructor term substitution θ is denoted $e\theta$ and defined as follows:

$$\begin{aligned}
\mathfrak{r}(t)\theta &\stackrel{\text{def}}{=} \theta(\mathfrak{Rfl})\\
\mathfrak{s}(e)\theta &\stackrel{\text{def}}{=} \theta(\mathfrak{Sym})\, e\theta\\
\mathfrak{t}(e_1, e_2)\theta &\stackrel{\text{def}}{=} \theta(\mathfrak{Trn})\, e_1\theta\, e_2\theta\\
\mathfrak{ba}(a^A.r, t)\theta &\stackrel{\text{def}}{=} \theta(\beta)\\
\mathfrak{bb}(u^{A[\Sigma_1]}.r, \Sigma_1.t)\theta &\stackrel{\text{def}}{=} \theta(\beta_\Box)\\
\mathfrak{ti}(\theta^w, \alpha)\theta &\stackrel{\text{def}}{=} \theta(\mathfrak{Trl})\\
\mathfrak{abC}(b^A.e)\theta &\stackrel{\text{def}}{=} \theta(\mathfrak{Abs})\, e\theta\\
\mathfrak{apC}(e_1, e_2)\theta &\stackrel{\text{def}}{=} \theta(\mathfrak{App})\, e_1\theta\, e_2\theta\\
\mathfrak{leC}(u^{A[\Sigma]}.e_1, e_2)\theta &\stackrel{\text{def}}{=} \theta(\mathfrak{Let})\, e_1\theta\, e_2\theta\\
\mathfrak{rpC}(\overline{e})\theta &\stackrel{\text{def}}{=} \theta(\mathfrak{Rpl})\, \overline{e}\theta
\end{aligned}$$

Likewise, the result of replacing each equivalence witness constructor in e with evidence substitution θ^w is denoted $e\theta^w$ and defined as follows:

$$\begin{aligned}
\mathfrak{r}(t)\theta^w &\stackrel{\text{def}}{=} \theta^w(\mathfrak{Rfl})\\
\mathfrak{s}(e)\theta^w &\stackrel{\text{def}}{=} \theta^w(\mathfrak{Sym}) \cdot e\theta^w\\
\mathfrak{t}(e_1, e_2)\theta^w &\stackrel{\text{def}}{=} \theta^w(\mathfrak{Trn}) \cdot e_1\theta^w \cdot e_2\theta^w\\
\mathfrak{ba}(a^A.r, t)\theta^w &\stackrel{\text{def}}{=} \theta^w(\beta)\\
\mathfrak{bb}(u^{A[\Sigma_1]}.r, \Sigma_1.t)\theta^w &\stackrel{\text{def}}{=} \theta^w(\beta_\Box)\\
\mathfrak{ti}(\theta^w, \alpha)\theta^w &\stackrel{\text{def}}{=} \theta^w(\mathfrak{Trl})\\
\mathfrak{abC}(b^A.e)\theta^w &\stackrel{\text{def}}{=} \theta^w(\mathfrak{Abs}) \cdot e\theta^w\\
\mathfrak{apC}(e_1, e_2)\theta^w &\stackrel{\text{def}}{=} \theta^w(\mathfrak{App}) \cdot e_1\theta^w \cdot e_2\theta^w\\
\mathfrak{leC}(u^{A[\Sigma]}.e_1, e_2)\theta^w &\stackrel{\text{def}}{=} \theta^w(\mathfrak{Let}) \cdot e_1\theta^w \cdot e_2\theta^w\\
\mathfrak{rpC}(\overline{e})\theta^w &\stackrel{\text{def}}{=} \theta^w(\mathfrak{Rpl}) \cdot \overline{e}\theta^w
\end{aligned}$$

C.2 Lemmata

Proof of Lemma 5.4.1 (Weakening).

1. If $\Delta; \Gamma; \Sigma \vdash M : A \mid s$ is derivable, then so is $\Delta'; \Gamma'; \Sigma' \vdash M : A \mid s$, where $\Delta \subseteq \Delta'$, $\Gamma \subseteq \Gamma'$ and $\Sigma \subseteq \Sigma'$.

2. If $\Delta; \Gamma; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e$ is derivable, then so is $\Delta'; \Gamma'; \Sigma' \vdash \mathsf{Eq}(A, s, t) \mid e$, where $\Delta \subseteq \Delta'$, $\Gamma \subseteq \Gamma'$ and $\Sigma \subseteq \Sigma'$.

Proof. By simultaneous induction on the derivation of $\Delta; \Gamma; \Sigma \vdash M : A \mid s$ and $\Delta; \Gamma; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e$. Note that it suffices to prove the result for

1. $\Delta_1, u : A[\Sigma], \Delta_2$ assuming $\Delta = \Delta_1, \Delta_2$;
2. $\Gamma_1, a : A, \Gamma_2$ assuming $\Gamma = \Gamma_1, \Gamma_2$; and
3. $\Sigma_1, \alpha : \mathsf{Eq}(A), \Sigma_2$ assuming $\Sigma = \Sigma_1, \Sigma_2$.

In what follows we write Δ^u for $\Delta_1, u : A[\Sigma], \Delta_2$ and similarly for the other two contexts.

- If the derivation ends in Var, the result is immediate.
- If the derivation ends in:

$$\frac{\Delta; \Gamma, b : A; \Sigma_1 \vdash M : B \mid s}{\Delta; \Gamma; \Sigma \vdash \lambda b : A.M : A \supset B \mid \lambda b : A.s} \supset \mathsf{I}$$

  by the IH $\Delta^u; \Gamma^a, b : A; \Sigma^\alpha \vdash M : B \mid s$. We conclude by resorting to $\supset \mathsf{I}$.
- Suppose the derivation ends in:

$$\frac{\Delta; \Gamma_1; \Sigma_1 \vdash M_1 : A \supset B \mid s_1 \quad \Delta; \Gamma_2; \Sigma_2 \vdash M_2 : A \mid s_2}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash M_1\, M_2 : B \mid s_1 \cdot s_2} \supset \mathsf{E}$$

  By the IH:
  – $\Delta^u; \Gamma_1^a; \Sigma_1^\alpha \vdash M_1 : A \supset B \mid s_1$
  – $\Delta^u; \Gamma_2; \Sigma_2 \vdash M_2 : A \mid s_2$

$$\frac{\begin{array}{c}\Delta^u; \Gamma_1^a; \Sigma_1^\alpha \vdash M_1 : A \supset B \mid s_1 \\ \Delta^u; \Gamma_2; \Sigma_2 \vdash M_2 : A \mid s_2\end{array}}{\Delta^u; \Gamma_{1,2}^a; \Sigma_{1,2}^\alpha \vdash M_1\, M_2 : B \mid s_1 \cdot s_2} \supset \mathsf{E}$$

- Suppose the derivation ends in TTLK:

$$\frac{\alpha : \mathsf{Eq}(A) \in \Sigma \quad \Delta; \cdot; \cdot \vdash \theta : \mathcal{T}^B \mid \theta^w}{\Delta; \Gamma; \Sigma \vdash \alpha\theta : B \mid \alpha\theta^w} \text{TTLK}$$

  We resort to the IH to obtain derivations of $\Delta^u; \cdot; \cdot \vdash \theta : \mathcal{T}^B \mid \theta^w$ and then conclude with an instance of TTLK.
- If the derivation ends in mVar the result is immediate.

$$\frac{u : A[\Sigma] \in \Delta \quad \Sigma\sigma \subseteq \Sigma_1}{\Delta; \Gamma_1; \Sigma_1 \vdash \langle u; \sigma \rangle : A \mid \langle u; \sigma \rangle} \text{TMVAR}$$

- Suppose the derivation ends in $\Box\mathsf{I}$:

$$\frac{\Delta; \cdot; \Sigma \vdash M : A \mid s \quad \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e}{\Delta; \Gamma_1; \Sigma_1 \vdash !_e^\Sigma M : [\![\Sigma.t]\!]A \mid \Sigma.t} \text{TBOX}$$

  We apply the IH to obtain:
  – $\Delta^u; \cdot; \Sigma \vdash M : A \mid s$; and
  – $\Delta^u; \cdot; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e$.
  Finally, we conclude by an instance of TBOX.
- Suppose $M = \mathsf{let}\, v : A[\Sigma] = M_1 \,\mathsf{in}\, M_2$ and $s = \text{LET}(v^{A[\Sigma]}.s_2, s_1)$ and the derivation ends in:

$$\frac{\begin{array}{c}\Delta; \Gamma_1; \Sigma_1 \vdash M_1 : [\![\Sigma.r]\!]A \mid s_1 \\ \Delta, v : A[\Sigma]; \Gamma_2; \Sigma_2 \vdash M_2 : C \mid s_2\end{array}}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash M : C_{\Sigma.r}^v \mid s} \Box\mathsf{E}$$

We resort to the IH and derive
- $\Delta^u; \Gamma_1^a; \Sigma_1^\alpha \vdash M_1 : [\![\Sigma.r]\!]A \mid s_1$
- $\Delta^u, v : A[\Sigma]; \Gamma_2; \Sigma_2 \vdash M_2 : C \mid s_2$

We conclude by an instance of $\Box\mathsf{E}$.

- If the derivation ends in:

$$\frac{\Delta; \Gamma_1; \Sigma_1 \vdash M : A \mid r \quad \Delta; \Gamma_1; \Sigma_1 \vdash \mathsf{Eq}(A, r, s) \mid e}{\Delta; \Gamma_1; \Sigma_1 \vdash e \rhd M : A \mid s} \mathsf{Eq}$$

We resort to the IH and derive
- $\Delta^u; \Gamma_1^a; \Sigma_1^\alpha \vdash M : A \mid r$
- $\Delta^u; \Gamma_1^a; \Sigma_1^\alpha \vdash \mathsf{Eq}(A, r, s) \mid e$

We conclude with an instance of Eq.

- If the derivation ends in:

$$\frac{\Delta; \Gamma_1; \Sigma_1 \vdash M : A \mid s}{\Delta; \Gamma_1; \Sigma_1 \vdash \mathsf{Eq}(A, s, s) \mid \mathfrak{r}(s)} \mathsf{EqRefl}$$

The result holds by the IH.

- If the derivation ends in:

$$\frac{\begin{array}{c}\Delta; \Gamma_1, b : A; \Sigma_1 \vdash M : B \mid r \\ \Delta; \Gamma_2; \Sigma_2 \vdash M' : A \mid s\end{array}}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash \mathsf{Eq}(B, r^b_s, (\lambda b : A.r) \cdot s) \mid e} \mathsf{Eq}\beta$$

By the IH.

- If $q \overset{\text{def}}{=} \textsc{let}(v^{A[\Sigma]}.r, \Sigma.s)$ and the derivation ends in:

$$\frac{\begin{array}{c}\Delta; \cdot; \Sigma \vdash A \mid t' \\ \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, t', s) \mid e \\ \Delta, v : A[\Sigma]; \Gamma_1; \Sigma_1 \vdash C \mid r \\ \Gamma_1 \subseteq \Gamma_2 \quad \Sigma_1 \subseteq \Sigma_2\end{array}}{\Delta; \Gamma_2; \Sigma_2 \vdash \mathsf{Eq}(C^v_{\Sigma.s}, r^v_{\Sigma.s}, q) \mid \mathfrak{bb}(v^{A[\Sigma]}.r, \Sigma.s)} \mathsf{Eq}\beta_\Box$$

By the IH we deduce:
- $\Delta^u; \cdot; \Sigma \vdash A \mid t'$
- $\Delta^u; \cdot; \Sigma \vdash \mathsf{Eq}(A, t', s) \mid e$
- $\Delta^u, v : A[\Sigma]; \Gamma_1; \Sigma_1 \vdash C \mid r$

We conclude with an instance of $\mathsf{Eq}\beta_\Box$.

- If the derivation ends in $\textsc{EqTlk}$:

$$\frac{\begin{array}{c}\Delta; \cdot; \Sigma_1 \vdash \mathsf{Eq}(A, s, t) \mid e \\ \Delta; \cdot; \cdot \vdash \mathcal{T}^B \mid \theta^w \\ \alpha : \mathsf{Eq}(A) \in \Sigma_2\end{array}}{\Delta; \Gamma; \Sigma_2 \vdash \mathsf{Eq}(B, e\theta^w, \alpha\theta^w) \mid \mathfrak{ti}(\theta^w, \alpha)} \textsc{EqTlk}$$

We resort to the IH to obtain a derivation of $\Delta^u; \cdot; \Sigma_1 \vdash \mathsf{Eq}(A, s, t) \mid e$ and $\Delta^u; \cdot; \cdot \vdash \mathcal{T}^B \mid \theta^w$. Finally, we apply $\textsc{EqTlk}$.

- If the derivation ends in EqSym or $\mathsf{EqTrans}$ we resort to the IH.
- If the derivation ends in:

$$\frac{\Delta; \Gamma_1, b : A; \Sigma_1 \vdash \mathsf{Eq}(B, s, r) \mid e}{\Delta; \Gamma_1; \Sigma_1 \vdash \mathsf{Eq}(A \supset B, \lambda b : A.s, \lambda b : A.r) \mid \mathfrak{abC}(b^A.e)} \mathsf{EqAbs}$$

By the IH.

- If the derivation ends in:

$$\frac{\begin{array}{c}\Delta;\Gamma_1;\Sigma_1 \vdash \mathsf{Eq}(A \supset B, s_1, s_2) \mid e_1 \\ \Delta;\Gamma_2;\Sigma_2 \vdash \mathsf{Eq}(A, t_1, t_2) \mid e_2\end{array}}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash \mathsf{Eq}(B, s_1 \cdot t_1, s_2 \cdot t_2) \mid \mathfrak{apC}(e_1, e_2)}\ \mathsf{EqApp}$$

By the IH.

- If $r_1 \overset{\text{def}}{=} \text{LET}(u^{A[\Sigma]}.t_1, s_1)$ and $r_2 \overset{\text{def}}{=} \text{LET}(u^{A[\Sigma]}.t_2, s_2)$ and the derivation ends in:

$$\frac{\begin{array}{c}\Delta;\Gamma_1;\Sigma_1 \vdash \mathsf{Eq}([\![\Sigma.r]\!]A, s_1, s_2) \mid e_1 \\ \Delta, u : A[\Sigma];\Gamma_2;\Sigma_2 \vdash \mathsf{Eq}(C, t_1, t_2) \mid e_2 \\ \Gamma_2 \subseteq \Gamma_3 \quad \Sigma_2 \subseteq \Sigma_3\end{array}}{\Delta;\Gamma_3;\Sigma_3 \vdash \mathsf{Eq}(C^u_{\Sigma.r}, r_1, r_2) \mid \mathfrak{leC}(u^{A[\Sigma]}.e_2, e_1)}\ \mathsf{EqLet}$$

By the IH.

- If the derivation ends in:

$$\frac{\begin{array}{c}\Delta;\cdot;\cdot \vdash \mathcal{T}^B(c_i) \mid \theta^w(c_i) \\ \Delta;\cdot;\cdot \vdash \mathsf{Eq}(\mathcal{T}^B(c_i), \theta^w(c_i), \theta'^w(c_i)) \mid e_i \\ \alpha : \mathsf{Eq}(A) \in \Sigma\end{array}}{\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(B, \alpha\theta'^w, \alpha\theta^w) \mid \mathfrak{rpC}(\overline{e})}\ \textsc{EqRpl}$$

By the IH we derive:
  - $\Delta^u;\cdot;\cdot \vdash \mathcal{T}^B(c_i) \mid \theta^w(c_i)$
  - $\Delta^u;\cdot;\cdot \vdash \mathsf{Eq}(\mathcal{T}^B(c_i), \theta^w(c_i), \theta'^w(c_i)) \mid e_i$

  We conclude with an instance of EqRpl.

▪

C.2.1 Proof of the Substitution Principle for Truth Hypothesis

Before proceeding we require the following auxiliary results.

Lemma C.2.1 1. If $\Delta;\Gamma;\Sigma \vdash M : A \mid s$, then $\mathsf{fvT}(s) \subseteq \mathsf{fvT}(M) \subseteq \mathsf{Dom}(\Gamma)$.
2. If $\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A, s, t) \mid e$, then $\mathsf{fvT}(s, t) \subseteq \mathsf{fvT}(e) \subseteq \mathsf{Dom}(\Gamma)$.

Proof. The proof is by simultaneous induction on $\Delta;\Gamma;\Sigma \vdash M : A \mid s$ and $\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A, s, t) \mid e$. The only interesting case for the former judgement is:

$$\frac{\Delta;\Gamma;\Sigma \vdash M : A \mid s \quad \Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A, s, t) \mid e}{\Delta;\Gamma;\Sigma \vdash e \rhd M : A \mid t}\ \textsc{TEq}$$

Here we use the IH w.r.t the latter judgement to deduce $\mathsf{fvT}(t) \subseteq \mathsf{fvT}(e) \subseteq \mathsf{fvT}(e \rhd M)$.

▪

Lemma C.2.2 (Strengthening) Suppose $a \in \mathsf{Dom}(\Gamma)$.

1. If $\Delta;\Gamma;\Sigma \vdash M : A \mid s$ and $a \notin \mathsf{fvT}(M)$, then $\Delta;\Gamma \setminus a;\Sigma \vdash M : A \mid s$.
2. If $\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A, s, t) \mid e$ and $a \notin \mathsf{fvT}(e)$, then $\Delta;\Gamma \setminus a;\Sigma \vdash \mathsf{Eq}(A, s, t) \mid e$.

Proof. By simultaneous induction on the derivations of $\Delta;\Gamma;\Sigma \vdash M : A \mid s$ and $\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(A, s, t) \mid e$.

▪

Lemma C.2.3 (Substitution Lemma)

1. $(s^{a_1}_{t_1})^{a_2}_{t_2} = (s^{a_2}_{t_2})^{a_1}_{t_1{}^{a_2}_{t_2}}$.

2. $(s^a_t)^u_{\Sigma.r} = (s^u_{\Sigma.r})^a_{t^u_{\Sigma.r}}$.

We now address the proof of the Substitution Principle for Truth Hypothesis.

Proof. By simultaneous induction on the derivations of $\Delta; \Gamma_1; \Sigma_1 \vdash M : B \mid s$ and $\Delta; \Gamma_1; \Sigma_1 \vdash \mathsf{Eq}(B, s_1, s_2) \mid e$. First note that:

- If $a \notin \mathsf{fvT}(M)$, then by Lem. C.2.1 also $a \notin \mathsf{fvT}(s)$. Therefore, $M^a_{N,t} = M$ and $s^a_t = s$ and the result follows from strengthening and weakening.
- If $a \notin \mathsf{fvT}(e)$, then by Lem. C.2.1 also $a \notin \mathsf{fvT}(s_1, s_2)$. Therefore, $e^a_t = e$, ${s_1}^a_t = s_1$ and ${s_2}^a_t = s_2$ and the result follows from strengthening and weakening.

So, in the sequel, we assume that $a \in \mathsf{fvT}(M, e)$.

- The derivation ends in:
$$\frac{b : A \in \Gamma}{\Delta; \Gamma_1; \Sigma_1 \vdash b : A \mid b}\,\mathsf{Var}$$
If $b = a$, then $A = B$ and we conclude by resorting to the hypothesis $\Delta; \Gamma_2; \Sigma_2 \vdash N : A \mid t$ and then weakening to obtain $\Delta; {\Gamma_1}^a_{\Gamma_2}; \Sigma_{1,2} \vdash N : A \mid t$. Otherwise, $\Delta; {\Gamma_1}^a_{\Gamma_2}; \Sigma_{1,2} \vdash b : A \mid b$ follows immediately from Var.
- If the derivation ends in:
$$\frac{\Delta; \Gamma_1, b : A; \Sigma_1 \vdash M : B \mid s}{\Delta; \Gamma_1; \Sigma_1 \vdash \lambda b : A.M : A \supset B \mid \lambda b : A.s}\supset \mathsf{I}$$
then $b \neq a$ and by the IH $\Delta; (\Gamma_1, b : A)^a_{\Gamma_2}; \Sigma_{1,2} \vdash M^a_{N,t} : B \mid s^a_t$. Since $(\Gamma_1, b : A)^a_{\Gamma_2} = {\Gamma_1}^a_{\Gamma_2}, b : A$, we conclude by resorting to $\supset \mathsf{I}$.
- Suppose the derivation ends in:
$$\frac{\begin{array}{c}\Delta; \Gamma_{11}; \Sigma_{11} \vdash M_1 : A \supset B \mid s_1 \\ \Delta; \Gamma_{12}; \Sigma_{12} \vdash M_2 : A \mid s_2\end{array}}{\Delta; \Gamma_{11,12}; \Sigma_{11,12} \vdash M_1\, M_2 : B \mid s_1 \cdot s_2}\supset \mathsf{E}$$
Since $\mathsf{fvT}(M_1) \cap \mathsf{fvT}(M_2) = \emptyset$, we consider three (mutually exclusive) subcases.
  - $a \in \mathsf{fvT}(M_1)$ (and, hence, $a \notin \mathsf{fvT}(s_2)$). By the IH $\Delta; {\Gamma_{11}}^a_{\Gamma_2}; \Sigma_{11,2} \vdash (M_1)^a_{N,t} : A \supset B \mid (s_1)^a_t$. We construct the derivation:
$$\frac{\begin{array}{c}\Delta; {\Gamma_{11}}^a_{\Gamma_2}; \Sigma_{11,2} \vdash (M_1)^a_{N,t} : A \supset B \mid (s_1)^a_t \\ \Delta; \Gamma_{12}; \Sigma_{12} \vdash M_2 : A \mid s_2\end{array}}{\Delta; {\Gamma_{11}}^a_{\Gamma_2}, \Gamma_{12}; \Sigma_{11,2,12} \vdash (M_1)^a_{N,t}\, M_2 : B \mid (s_1)^a_t \cdot s_2}\supset \mathsf{E}$$
  - $a \in \mathsf{fvT}(M_2)$ (and, hence, $a \notin \mathsf{fvT}(s_1)$). By the IH $\Delta; {\Gamma_{12}}^a_{\Gamma_2}; \Sigma_{12,2} \vdash (M_2)^a_{N,t} : A \mid (s_2)^a_t$. We construct the derivation:
$$\frac{\begin{array}{c}\Delta; \Gamma_{11}; \Sigma_{11} \vdash M_1 : A \supset B \mid s_1 \\ \Delta; {\Gamma_{12}}^a_{\Gamma_2}; \Sigma_{12,2} \vdash (M_2)^a_{N,t} : A \mid (s_2)^a_t\end{array}}{\Delta; \Gamma_{11}, {\Gamma_{12}}^a_{\Gamma_2}; \Sigma_{11,12,2} \vdash M_1\, (M_2)^a_{N,t} : B \mid s_1 \cdot (s_2)^a_t}\supset \mathsf{E}$$
  - $a \notin \mathsf{fvT}(M_1\, M_2)$. This case has already been dealt with.
- Suppose the derivation ends in mVar, $\Box\mathsf{I}$ or TTLK, then $a \notin \mathsf{fvT}(M)$ and these cases have already been dealt with.
- Suppose $M = \mathsf{let}\, u : A[\Sigma] = M_1\, \mathsf{in}\, M_2$ and the derivation ends in:
$$\frac{\begin{array}{c}\Delta; \Gamma_{11}; \Sigma_{11} \vdash M : [\![\Sigma.r]\!]A \mid s_1 \\ \Delta, u : A[\Sigma]; \Gamma_{12}; \Sigma_{12} \vdash N : C \mid s_2\end{array}}{\Delta; \Gamma_{11,12}; \Sigma_{11,12} \vdash M : C^u_{\Sigma.r} \mid \mathrm{LET}(u^{A[\Sigma]}.s_2, s_1)}\Box\mathsf{E}$$
Since $\mathsf{fvT}(M_1) \cap \mathsf{fvT}(M_2) = \emptyset$, we consider three (mutually exclusive) subcases.

– $a \in \mathsf{fvT}(M_1)$ (hence $a \notin \mathsf{fvT}(s_2)$). Then by the IH, $\Delta; {\Gamma_{11}}^a_{\Gamma_2}; \Sigma_{11,2} \vdash (M_1)^a_{N,t} : [\![\Sigma.r]\!]A \mid (s_1)^a_t$. We construct the following derivation where $M' = \mathsf{let}\, u : A[\Sigma] = (M_1)^a_{N,t} \,\mathsf{in}\, M_2$:

$$\frac{\Delta; {\Gamma_{11}}^a_{\Gamma_2}; \Sigma_{11,2} \vdash (M_1)^a_{N,t} : [\![\Sigma.r]\!]A \mid (s_1)^a_t \quad \Delta, u : A[\Sigma]; \Gamma_{12}; \Sigma_{12} \vdash M_2 : C \mid s_2}{\Delta; {\Gamma_{11}}^a_{\Gamma_2}, \Gamma_{12}; \Sigma_{11,2,12} \vdash M' : C^u_{\Sigma.r} \mid \mathrm{LET}(u^{A[\Sigma]}.s_2, (s_1)^a_t)} \Box\mathsf{E}$$

– $a \in \mathsf{fvT}(M_2)$ (hence $a \notin \mathsf{fvT}(s_1)$). Then by the IH, $\Delta, u : A[\Sigma]; {\Gamma_{12}}^a_{\Gamma_2}; \Sigma_{12,2} \vdash (M_2)^a_{N,t} : C \mid (s_2)^a_t$. We construct the following derivation where $M' = \mathsf{let}\, u : A[\Sigma] = M_1 \,\mathsf{in}\, (M_2)^a_{N,t}$:

$$\frac{\Delta; \Gamma_{11}; \Sigma_{11} \vdash M_1 : [\![\Sigma.r]\!]A \mid s_1 \quad \Delta, u : A[\Sigma]; {\Gamma_{12}}^a_{\Gamma_2}; \Sigma_{12,2} \vdash (M_2)^a_{N,t} : C \mid (s_2)^a_t}{\Delta; \Gamma_{11}, {\Gamma_{12}}^a_{\Gamma_2}; \Sigma_{11,12,2} \vdash M' : C^u_{\Sigma.r} \mid \mathrm{LET}(u^{A[\Sigma]}.(s_2)^a_t, s_1)} \Box\mathsf{E}$$

– $a \notin \mathsf{fvT}(M)$. This case has already been dealt with.

- If the derivation ends in:

$$\frac{\Delta; \Gamma_1; \Sigma_1 \vdash M : A \mid r \quad \Delta; \Gamma_1; \Sigma_1 \vdash \mathsf{Eq}(A, r, s) \mid e}{\Delta; \Gamma_1; \Sigma_1 \vdash e \rhd M : A \mid s} \mathsf{Eq}$$

  Then by the IH:
  – $\Delta; {\Gamma_1}^a_{\Gamma_2}; \Sigma_{1,2} \vdash M^a_{N,t} : A \mid r^a_t$.
  – $\Delta; {\Gamma_1}^a_{\Gamma_2}; \Sigma_{1,2} \vdash \mathsf{Eq}(A, r^a_t, s^a_t) \mid e^a_t$.
  We conclude by:

$$\frac{\Delta; {\Gamma_1}^a_{\Gamma_2}; \Sigma_{1,2} \vdash M^a_{N,t} : A \mid r^a_t \quad \Delta; {\Gamma_1}^a_{\Gamma_2}; \Sigma_{1,2} \vdash \mathsf{Eq}(A, r^a_t, s^a_t) \mid e^a_t}{\Delta; {\Gamma_1}^a_{\Gamma_2}; \Sigma_{1,2} \vdash e^a_t \rhd M^a_{N,t} : A \mid s^a_t} \mathsf{Eq}$$

- If the derivation ends in:

$$\frac{\Delta; \Gamma_1; \Sigma_1 \vdash M : A \mid s}{\Delta; \Gamma_1; \Sigma_1 \vdash \mathsf{Eq}(A, s, s) \mid \mathfrak{r}(s)} \mathsf{EqRefl}$$

  By the IH $\Delta; {\Gamma_1}^a_{\Gamma_2}; \Sigma_{1,2} \vdash M^a_{N,t} : A \mid s^a_t$. Therefore we obtain

$$\frac{\Delta; {\Gamma_1}^a_{\Gamma_2}; \Sigma_{1,2} \vdash M^a_{N,t} : A \mid s^a_t}{\Delta; {\Gamma_1}^a_{\Gamma_2}; \Sigma_{1,2} \vdash \mathsf{Eq}(A, s^a_t, s^a_t) \mid \mathfrak{r}(s^a_t)} \mathsf{EqRefl}$$

- If $e = \mathfrak{ba}(b^A.r, s)$ and the derivation ends in:

$$\frac{\Delta; \Gamma_{11}, b : A; \Sigma_{11} \vdash M_1 : B \mid r \quad \Delta; \Gamma_{12}; \Sigma_{12} \vdash M_2 : A \mid s}{\Delta; \Gamma_{11,12}; \Sigma_{11,12} \vdash \mathsf{Eq}(B, r^b_s, (\lambda b : A.r) \cdot s) \mid e} \mathsf{Eq}\beta$$

  By Lem. C.2.1 $\mathsf{fvT}(M_1) \cap \mathsf{fvT}(M_2) = \emptyset$. We therefore have two subcases.
  – $a \in \mathsf{fvT}(M_1)$. Then by the IH, $\Delta; (\Gamma_{11}, b : A)^a_{\Gamma_2}; \Sigma_{11,2} \vdash (M_1)^a_{N,t} : B \mid r^a_t$. We construct the following derivation where $e' = \mathfrak{ba}(b^A.r^a_t, s)$:

$$\frac{\Delta; {\Gamma_{11}}^a_{\Gamma_2}, b : A; \Sigma_{11,2} \vdash (M_1)^a_{N,t} : B \mid r^a_t \quad \Delta; \Gamma_{12}; \Sigma_{12} \vdash M_2 : A \mid s}{\Delta; {\Gamma_{11}}^a_{\Gamma_2}, \Gamma_{12}; \Sigma_{11,2,12} \vdash \mathsf{Eq}(B, (r^a_t)^b_s, (\lambda b : A.r^a_t) \cdot s) \mid e'} \mathsf{Eq}\beta$$

  – $a \in \mathsf{fvT}(M_2)$. Then by the IH, $\Delta; {\Gamma_{12}}^a_{\Gamma_2}; \Sigma_{12,2} \vdash (M_2)^a_{N,t} : A \mid s^a_t$. We construct the following derivation where $e' = \mathfrak{ba}(b^A.r, s^a_t)$:

$$\frac{\Delta; \Gamma_{11}, b : A; \Sigma_{11} \vdash M_1 : B \mid r \quad \Delta; {\Gamma_{12}}^a_{\Gamma_2}; \Sigma_{12,2} \vdash (M_2)^a_{N,t} : A \mid s^a_t}{\Delta; \Gamma_{11}, {\Gamma_{12}}^a_{\Gamma_2}; \Sigma_{11,12,2} \vdash \mathsf{Eq}(B, r^b_{s^a_t}, (\lambda b : A.r) \cdot s^a_t) \mid e'} \mathsf{Eq}\beta$$

- If the derivation ends in:

$$\frac{\begin{array}{c}\Delta; \cdot; \Sigma \vdash A \mid t' \\ \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, t', s) \mid e \\ \Delta, u : A[\Sigma]; \Gamma_{11}; \Sigma_{11} \vdash C \mid r \\ \Gamma_{11} \subseteq \Gamma_1 \quad \Sigma_{11} \subseteq \Sigma_1\end{array}}{\Delta; \Gamma_1; \Sigma_1 \vdash \mathsf{Eq}(C^u_{\Sigma.s}, r^u_{\Sigma.s}, \textsc{let}(u^{A[\Sigma]}.r, \Sigma.s)) \mid \mathfrak{bb}(u^{A[\Sigma]}.r, \Sigma.s)} \mathsf{Eq}\beta_\Box$$

Then by the IH, $\Delta, u : A[\Sigma]; \Gamma_{11}{}^a_{\Gamma_2}; \Sigma_{11,2} \vdash C \mid r^a_t$. We construct the following derivation where $q = \textsc{let}(u^{A[\Sigma]}.r^a_t, \Sigma.s)$:

$$\frac{\begin{array}{c}\Delta; \cdot; \Sigma \vdash A \mid t' \\ \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, t', s) \mid e \\ \Delta, u : A[\Sigma]; \Gamma_{11}{}^a_{\Gamma_2}; \Sigma_{11,2} \vdash C \mid r^a_t\end{array}}{\Delta; \Gamma_1{}^a_{\Gamma_2}; \Sigma_{1,2} \vdash \mathsf{Eq}(C^u_{\Sigma.s}, (r^a_t)^u_{\Sigma.s}, q) \mid \mathfrak{bb}(u^{A[\Sigma]}.r^a_t, \Sigma.s)} \mathsf{Eq}\beta_\Box$$

- The case in which the derivation ends in EqTlk has already been dealt with since in this case $a \notin \mathsf{fvT}(e)$.
- Suppose the derivation ends in:

$$\frac{\Delta; \Gamma_1; \Sigma_1 \vdash \mathsf{Eq}(A, s_1, s_2) \mid e}{\Delta; \Gamma_1; \Sigma_1 \vdash \mathsf{Eq}(A, s_2, s_1) \mid \mathfrak{s}(e)} \mathsf{EqSym}$$

By the IH we have $\Delta; \Gamma_1{}^a_{\Gamma_2}; \Sigma_{1,2} \vdash \mathsf{Eq}(A, (s_1)^a_t, (s_2)^a_t) \mid e^a_t$. We conclude by resorting to EqSym.
- If the derivation ends in:

$$\frac{\begin{array}{c}\Delta; \Gamma_1; \Sigma_1 \vdash \mathsf{Eq}(A, s_1, s_2) \mid e_1 \\ \Delta; \Gamma_1; \Sigma_1 \vdash \mathsf{Eq}(A, s_2, s_3) \mid e_2\end{array}}{\Delta; \Gamma_1; \Sigma_1 \vdash \mathsf{Eq}(A, s_1, s_3) \mid \mathfrak{t}(e_1, e_2)} \mathsf{EqTrans}$$

we resort to the IH.
- If the derivation ends in:

$$\frac{\Delta; \Gamma_1, b : A; \Sigma_1 \vdash \mathsf{Eq}(B, s, r) \mid e}{\Delta; \Gamma_1; \Sigma_1 \vdash \mathsf{Eq}(A \supset B, \lambda b : A.s, \lambda b : A.r) \mid \mathfrak{abC}(b^A.e)} \mathsf{EqAbs}$$

Then $b \neq a$ and by the IH $\Delta; (\Gamma_1, b : A)^a_{\Gamma_2}; \Sigma_{1,2} \vdash \mathsf{Eq}(B, s^a_t, r^a_t) \mid e^a_t$. We conclude by resorting to EqAbs.
- Suppose $e = \mathfrak{apC}(e_1, e_2)$ and the derivation ends in:

$$\frac{\begin{array}{c}\Delta; \Gamma_{11}; \Sigma_{11} \vdash \mathsf{Eq}(A \supset B, s_1, s_2) \mid e_1 \\ \Delta; \Gamma_{12}; \Sigma_{12} \vdash \mathsf{Eq}(A, t_1, t_2) \mid e_2\end{array}}{\Delta; \Gamma_{11,12}; \Sigma_{11,12} \vdash \mathsf{Eq}(B, s_1 \cdot t_1, s_2 \cdot t_2) \mid e} \mathsf{EqApp}$$

Note that $\mathsf{fvT}(e_1) \cap \mathsf{fvT}(e_2) = \emptyset$. We consider two (mutually exclusive) subcases:
1. $a \in \mathsf{fvT}(e_1)$ (hence $a \notin \mathsf{fvT}(e_2)$). Then by the IH, $\Delta; \Gamma_{11}{}^a_{\Gamma_2}; \Sigma_{11,2} \vdash \mathsf{Eq}(A \supset B, s_1{}^a_t, s_2{}^a_t) \mid e_1{}^a_t$. We construct the derivation:
$$\frac{\begin{array}{c}\Delta; \Gamma_{11}{}^a_{\Gamma_2}; \Sigma_{11,2} \vdash \mathsf{Eq}(A \supset B, s_1{}^a_t, s_2{}^a_t) \mid e_1{}^a_t \\ \Delta; \Gamma_{12}; \Sigma_{12} \vdash \mathsf{Eq}(A, t_1, t_2) \mid e_2\end{array}}{\Delta; \Gamma_{11}{}^a_{\Gamma_2}, \Gamma_{12}; \Sigma_{11,2,12} \vdash \mathsf{Eq}(B, s_1{}^a_t \cdot t_1, s_2{}^a_t \cdot t_2) \mid \mathfrak{apC}(e_1{}^a_t, e_2)} \mathsf{EqApp}$$
2. $a \in \mathsf{fvT}(e_2)$ (hence $a \notin \mathsf{fvT}(e_1)$). Then by the IH, $\Delta; \Gamma_{12}{}^a_{\Gamma_2}; \Sigma_{12,2} \vdash \mathsf{Eq}(A, t_1{}^a_t, t_2{}^a_t) \mid e_2{}^a_t$. We construct the derivation:
$$\frac{\begin{array}{c}\Delta; \Gamma_{11}; \Sigma_{11} \vdash \mathsf{Eq}(A \supset B, s_1, s_2) \mid e_1 \\ \Delta; \Gamma_{12}{}^a_{\Gamma_2}; \Sigma_{12,2} \vdash \mathsf{Eq}(A, t_1{}^a_t, t_2{}^a_t) \mid e_2{}^a_t\end{array}}{\Delta; \Gamma_{11}, \Gamma_{12}{}^a_{\Gamma_2}; \Sigma_{11,12,2} \vdash \mathsf{Eq}(B, s_1 \cdot t_1{}^a_t, s_2 \cdot t_2{}^a_t) \mid \mathfrak{apC}(e_1, e_2{}^a_t)} \mathsf{EqApp}$$

- Suppose the derivation ends in:

$$\dfrac{\begin{array}{c}\Delta;\Gamma_{11};\Sigma_{11}\vdash \mathsf{Eq}([\![\Sigma.r]\!]A,s_1,s_2)\mid e_1\\ \Delta,u:A[\Sigma];\Gamma_{12};\Sigma_{12}\vdash \mathsf{Eq}(C,t_1,t_2)\mid e_2\end{array}}{\Delta;\Gamma_{11,12};\Sigma_{11,12}\vdash \mathsf{Eq}(C^{u}_{\Sigma.r},r_1,r_2)\mid \mathfrak{leC}(u^{A[\Sigma]}.e_2,e_1)}\ \mathsf{EqLet}$$

where $r_1=\textsc{let}(u^{A[\Sigma]}.t_1,s_1)$ and $r_2=\textsc{let}(u^{A[\Sigma]}.t_2,s_2)$. Note that $\mathsf{fvT}(e_1)\cap\mathsf{fvT}(e_2)=\emptyset$. We consider two (mutually exclusive) subcases:

1. $a\in\mathsf{fvT}(e_1)$. Then by the IH, $\Delta;{\Gamma_{11}}^{a}_{\Gamma_2};\Sigma_{11,2}\vdash\mathsf{Eq}([\![\Sigma.r]\!]A,{s_1}^{a}_{t},{s_2}^{a}_{t})\mid {e_1}^{a}_{t}$. We resort to EqLet and construct the following derivation where $r_1'=\textsc{let}(u^{A[\Sigma]}.t_1,{s_1}^{a}_{t})$ and $r_2'=\textsc{let}(u^{A[\Sigma]}.t_2,{s_2}^{a}_{t})$ and $e'\overset{\text{def}}{=}\mathfrak{leC}(u^{A[\Sigma]}.e_2,{e_1}^{a}_{t})$:

$$\dfrac{\begin{array}{c}\Delta;{\Gamma_{11}}^{a}_{\Gamma_2};\Sigma_{11,2}\vdash \mathsf{Eq}([\![\Sigma.r]\!]A,{s_1}^{a}_{t},{s_2}^{a}_{t})\mid {e_1}^{a}_{t}\\ \Delta,u:A[\Sigma];\Gamma_{12};\Sigma_{12}\vdash \mathsf{Eq}(C,t_1,t_2)\mid e_2\end{array}}{\Delta;{\Gamma_{11}}^{a}_{\Gamma_2},\Gamma_{12};\Sigma_{11,2,12}\vdash \mathsf{Eq}(C^{u}_{\Sigma.r},r_1',r_2')\mid e'}\ \mathsf{EqLet}$$

2. $a\in\mathsf{fvT}(e_2)$. Then by the IH, $\Delta,u:A[\Sigma];{\Gamma_{12}}^{a}_{\Gamma_2};\Sigma_{12,2}\vdash\mathsf{Eq}([\![\Sigma.r]\!]A,{t_1}^{a}_{t},{t_2}^{a}_{t})\mid {e_2}^{a}_{t}$. We resort to EqLet and construct the following derivation where $r_1'=\textsc{let}(u^{A[\Sigma]}.{t_1}^{a}_{t},s_1)$ and $r_2'=\textsc{let}(u^{A[\Sigma]}.{t_2}^{a}_{t},s_2)$ and $e'\overset{\text{def}}{=}\mathfrak{leC}(u^{A[\Sigma]}.{e_2}^{a}_{t},e_1)$:

$$\dfrac{\begin{array}{c}\Delta;\Gamma_{11};\Sigma_{11}\vdash \mathsf{Eq}([\![\Sigma.r]\!]A,{s_1}^{a}_{t},{s_2}^{a}_{t})\mid {e_1}^{a}_{t}\\ \Delta,u:A[\Sigma];{\Gamma_{12}}^{a}_{\Gamma_2};\Sigma_{12,2}\vdash \mathsf{Eq}([\![\Sigma.r]\!]A,{t_1}^{a}_{t},{t_2}^{a}_{t})\mid {e_2}^{a}_{t}\end{array}}{\Delta;\Gamma_{11},{\Gamma_{12}}^{a}_{\Gamma_2};\Sigma_{11,12,2}\vdash \mathsf{Eq}(C^{u}_{\Sigma.r},r_1',r_2')\mid e'}\ \mathsf{EqLet}$$

- If the derivation ends in:

$$\dfrac{\begin{array}{c}\Delta;\cdot;\cdot\vdash \mathsf{Eq}(\mathcal{T}^{B},\theta'^{w},\theta^{w})\mid e_i\\ \alpha:\mathsf{Eq}(A)\in\Sigma_1\end{array}}{\Delta;\Gamma_1;\Sigma_1\vdash \mathsf{Eq}(B,\alpha\theta'^{w},\alpha\theta^{w})\mid \mathfrak{rpC}(\overline{e})}\ \textsc{EqRpl}$$

The result is immediate by an application of EqRpl.

∎

C.2.2 Proof of the Substitution Principle for Validity Hypothesis

The proof requires the property that trail variable renaming preserve typability. In the turn, this property requires the following lemma which is proved by induction on s.

Lemma C.2.4

1. $(s^{a}_{t})\sigma=(s\sigma)^{a}_{t\sigma}$.
2. $(s^{u}_{\Sigma.t})\sigma=(s\sigma)^{u}_{\Sigma.t}$.
3. $(s^{u_1}_{\Sigma_1.t_1})^{u_2}_{\Sigma_2.t_2}=(s^{u_2}_{\Sigma_2.t_2})^{u_1}_{\Sigma_1.{t_1}^{u_2}_{\Sigma_2.t_2}}$.

Proof. Induction on s. ∎

Lemma C.2.5 Let σ be a renaming s.t. $\mathsf{Dom}(\Sigma_1)\subseteq\mathsf{Dom}(\sigma)$.

1. If $\Delta;\Gamma;\Sigma_1\vdash M:A\mid t$ is derivable, then so is $\Delta;\Gamma;\sigma(\Sigma_1)\vdash M\sigma:A\mid t\sigma$.
2. If $\Delta;\Gamma;\Sigma_1\vdash\mathsf{Eq}(A,s,t)\mid e$ is derivable, then so is $\Delta;\Gamma;\sigma(\Sigma_1)\vdash\mathsf{Eq}(A,\sigma(s),\sigma(t))\mid\sigma(e)$.

Proof. By induction on the derivation of $\Delta;\Gamma;\Sigma_1\vdash M:A\mid t$.

- The derivation ends in:

$$\dfrac{b:A\in\Gamma}{\Delta;\Gamma;\Sigma_1\vdash b:A\mid b}\ \mathsf{Var}$$

Then $\Delta; \Gamma; \sigma(\Sigma_1) \vdash b : A \mid b$ is immediate.

- If the derivation ends in:

$$\frac{\Delta; \Gamma, b : A; \Sigma_1 \vdash M : B \mid r_1}{\Delta; \Gamma; \Sigma_1 \vdash \lambda b : A.M : A \supset B \mid \lambda b : A.r_1} \text{TAbs}$$

Then by the IH $\Delta; \Gamma, b : A; \sigma(\Sigma_1) \vdash M\sigma : B \mid r_1\sigma$. We construct the derivation:

$$\frac{\Delta; \Gamma, b : A; \sigma(\Sigma_1) \vdash M\sigma : B \mid r_1\sigma}{\Delta; \Gamma; \sigma(\Sigma_1) \vdash \lambda b : A.M\sigma : A \supset B \mid \lambda b : A.r_1\sigma} \text{TAbs}$$

By the definition of renaming, $\Delta; \Gamma; \sigma(\Sigma_1) \vdash (\lambda b : A.M)\sigma : A \supset B \mid (\lambda b : A.r_1)\sigma$.

- Suppose the derivation ends in:

$$\frac{\begin{array}{c}\Delta; \Gamma_1; \Sigma_{11} \vdash M_1 : A \supset B \mid s_1 \\ \Delta; \Gamma_2; \Sigma_{12} \vdash M_2 : A \mid s_2\end{array}}{\Delta; \Gamma_{1,2}; \Sigma_{11,12} \vdash M_1\, M_2 : B \mid s_1 \cdot s_2} \text{TApp}$$

By the IH:
– $\Delta; \Gamma_1; \sigma(\Sigma_{11}) \vdash M_1\sigma : A \supset B \mid s_1\sigma$
– $\Delta; \Gamma_2; \sigma(\Sigma_{12}) \vdash M_2\sigma : A \mid s_2\sigma$

We construct the derivation:

$$\frac{\begin{array}{c}\Delta; \Gamma_1; \sigma(\Sigma_{11}) \vdash M_1\sigma : A \supset B \mid s_1\sigma \\ \Delta; \Gamma_2; \sigma(\Sigma_{12}) \vdash M_2\sigma : A \mid s_2\sigma\end{array}}{\Delta; \Gamma_{1,2}; \sigma(\Sigma_{11,12}) \vdash M_1\sigma\, M_2\sigma : B \mid s_1\sigma \cdot s_2\sigma} \text{TApp}$$

- If the derivation ends in:

$$\frac{u : A[\Sigma] \in \Delta \quad \Sigma\sigma' \subseteq \Sigma_1}{\Delta; \Gamma; \Sigma_1 \vdash \langle u; \sigma' \rangle : A \mid \langle u; \sigma' \rangle} \text{TMVar}$$

Then

$$\frac{u : A[\Sigma] \in \Delta \quad \Sigma(\sigma \circ \sigma') \subseteq \sigma(\Sigma_1)}{\Delta; \Gamma; \sigma(\Sigma_1) \vdash \langle u; \sigma \circ \sigma' \rangle : A \mid \langle u; \sigma \circ \sigma' \rangle} \text{TMVar}$$

- If the derivation ends in:

$$\frac{\Delta; \cdot; \Sigma \vdash M : A \mid s \quad \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e}{\Delta; \Gamma; \Sigma_1 \vdash !^{\Sigma_1}_{e} M : [\![\Sigma_1.t]\!]A \mid \Sigma_1.t} \text{TBox}$$

The result is trivial. Note that $(!^{\Sigma}_{e} M)\sigma = !^{\Sigma}_{e} M$ and $(\Sigma_1.t)\sigma = \Sigma_1.t$.

- Suppose the derivation ends in:

$$\frac{\begin{array}{c}\Delta; \Gamma_1; \Sigma_{11} \vdash M : [\![\Sigma.r]\!]A \mid s_1 \\ \Delta, u : A[\Sigma]; \Gamma_2; \Sigma_{12} \vdash N : C \mid s_2\end{array}}{\begin{array}{c}\Delta; \Gamma_{1,2}; \Sigma_{11,12} \vdash \mathsf{let}\, u : A[\Sigma] = M \,\mathsf{in}\, N : C^{u}_{\Sigma.r} \mid \\ \textsc{let}(u^{A[\Sigma]}.s_2, s_1)\end{array}} \text{TLetB}$$

By the IH:
– $\Delta; \Gamma_1; \sigma(\Sigma_{11}) \vdash M\sigma : [\![\Sigma.r]\!]A \mid s_1\sigma$
– $\Delta, u : A[\Sigma]; \Gamma_2; \sigma(\Sigma_{12}) \vdash N\sigma : C \mid s_2\sigma$

We construct the derivation:

$$\dfrac{\begin{array}{c}\Delta;\Gamma_1;\sigma(\Sigma_{11}) \vdash M\sigma : [\![\Sigma.r]\!]A \mid s_1\sigma \\ \Delta, u : A[\Sigma];\Gamma_2;\sigma(\Sigma_{12}) \vdash N\sigma : C \mid s_2\sigma\end{array}}{\begin{array}{c}\Delta;\Gamma_{1,2};\sigma(\Sigma_{11,12}) \vdash \mathsf{let}\, u : A[\Sigma] = M\sigma \,\mathsf{in}\, N\sigma : C^u_{\Sigma.r} \mid \\ \text{LET}(u^{A[\Sigma]}.s_2\sigma, s_1\sigma)\end{array}}\ \text{TLetB}$$

- The result holds immediately if the derivation is:

$$\dfrac{\alpha : \mathsf{Eq}(A) \in \Sigma \quad \Delta;\cdot;\cdot \vdash \theta : \mathcal{T}^B \mid \theta^w}{\Delta;\Gamma;\Sigma_1 \vdash \alpha\theta : B \mid \alpha\theta^w}\ \text{TTlk}$$

  since $(\alpha\theta)\sigma = \alpha\theta$ and we can resort to TTlk.
- If the derivation ends in:

$$\dfrac{\begin{array}{c}\Delta;\Gamma;\Sigma_1 \vdash M : A \mid r \\ \Delta;\Gamma;\Sigma_1 \vdash \mathsf{Eq}(A,r,s) \mid e\end{array}}{\Delta;\Gamma;\Sigma_1 \vdash e \rhd M : A \mid s}\ \text{TEq}$$

  Then by IH $\Delta;\Gamma;\sigma(\Sigma_1) \vdash M\sigma : A \mid r\sigma$ and $\Delta;\Gamma;\sigma(\Sigma_1) \vdash \mathsf{Eq}(A, r\sigma, s\sigma) \mid e\sigma$.
  We construct the derivation:

$$\dfrac{\begin{array}{c}\Delta;\Gamma;\sigma(\Sigma_1) \vdash M\sigma : A \mid r\sigma \\ \Delta;\Gamma;\sigma(\Sigma_1) \vdash \mathsf{Eq}(A,r\sigma,s\sigma) \mid e\sigma\end{array}}{\Delta;\Gamma;\sigma(\Sigma_1) \vdash e\sigma \rhd M\sigma : A \mid s\sigma}\ \text{TEq}$$

- If the derivation ends in:

$$\dfrac{\Delta;\Gamma;\Sigma_1 \vdash M : A \mid s}{\Delta;\Gamma;\Sigma_1 \vdash \mathsf{Eq}(A,s,s) \mid \mathfrak{r}(s)}\ \mathsf{EqRefl}$$

  Then by IH $\Delta;\Gamma;\sigma(\Sigma_1) \vdash M\sigma : A \mid s\sigma$.
  We construct the derivation:

$$\dfrac{\Delta;\Gamma;\sigma(\Sigma_1) \vdash M\sigma : A \mid s\sigma}{\Delta;\Gamma;\sigma(\Sigma_1) \vdash \mathsf{Eq}(A,s\sigma,s\sigma) \mid \mathfrak{r}(s\sigma)}\ \mathsf{EqRefl}$$

- If the derivation ends in:

$$\dfrac{\begin{array}{c}\Delta;\Gamma_1, b : A;\Sigma_{11} \vdash M : B \mid r \\ \Delta;\Gamma_2;\Sigma_{12} \vdash N : A \mid s\end{array}}{\Delta;\Gamma_{1,2};\Sigma_{11,12} \vdash \mathsf{Eq}(B, r^b_s, (\lambda b : A.r)\cdot s) \mid \mathfrak{ba}(b^A.r, s)}\ \mathsf{Eq}\beta$$

  By the IH:
  - $\Delta;\Gamma_1, b : A;\sigma(\Sigma_{11}) \vdash M\sigma : B \mid r\sigma$
  - $\Delta;\Gamma_2;\sigma(\Sigma_{12}) \vdash N\sigma : A \mid s\sigma$

  We construct the derivation and conclude from Lem. C.2.5(1):

$$\dfrac{\begin{array}{c}\Delta;\Gamma_1, b : A;\sigma(\Sigma_{11}) \vdash M\sigma : B \mid r\sigma \\ \Delta;\Gamma_2;\sigma(\Sigma_{12}) \vdash N\sigma : A \mid s\sigma\end{array}}{\begin{array}{c}\Delta;\Gamma_{1,2};\sigma(\Sigma_{11,12}) \vdash \mathsf{Eq}(B, r\sigma^b_{s\sigma}, (\lambda b : A.r\sigma)\cdot s\sigma) \mid \\ \mathfrak{ba}(b^A.r\sigma, s)\end{array}}\ \mathsf{Eq}\beta$$

- If the derivation ends in:

$$\dfrac{\begin{array}{c}\Delta;\cdot;\Sigma \vdash A \mid r \\ \Delta;\cdot;\Sigma \vdash \mathsf{Eq}(A,r,s) \mid e \\ \Delta, u : A[\Sigma];\Gamma';\Sigma' \vdash C \mid t \\ \Gamma' \subseteq \Gamma \quad \Sigma' \subseteq \Sigma\end{array}}{\begin{array}{c}\Delta;\Gamma;\Sigma \vdash \mathsf{Eq}(C^u_{\Sigma.s}, t^u_{\Sigma.s}, \text{LET}(u^{A[\Sigma]}.t, \Sigma.s)) \mid \\ \mathfrak{bb}(u^{A[\Sigma]}.t, \Sigma.s)\end{array}}\ \mathsf{Eq}\beta_{\Box}$$

We resort to the IH to deduce $\Delta, u : A[\Sigma]; \Gamma; \sigma(\Sigma_1) \vdash C \mid \sigma(t)$ and then derive:

$$\frac{\begin{array}{c}\Delta; \cdot; \Sigma \vdash A \mid r \\ \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, r, s) \mid e \\ \Delta, u : A[\Sigma]; \Gamma; \sigma(\Sigma_1) \vdash C \mid \sigma(t)\end{array}}{\begin{array}{c}\Delta; \Gamma; \sigma(\Sigma_1) \vdash \mathsf{Eq}(C^u_{\Sigma.s}, \sigma(t)^u_{\Sigma.s}, \text{LET}(u^{A[\Sigma]}.\sigma(t), \Sigma.s)) \mid \\ \mathfrak{bb}(u^{A[\Sigma]}.\sigma(t), \Sigma.s)\end{array}} \mathsf{Eq}\beta_\Box$$

- The case in which the derivation ends in:

$$\frac{\begin{array}{c}\Delta; \cdot; \Sigma_1' \vdash \mathsf{Eq}(A, s, t) \mid e \\ \Delta; \cdot; \cdot \vdash \mathcal{T}^B \mid \theta^w \\ \alpha : \mathsf{Eq}(A) \in \Sigma_1\end{array}}{\Delta; \Gamma; \Sigma_1 \vdash \mathsf{Eq}(B, e\theta^w, \alpha\theta^w) \mid \mathfrak{ti}(\theta^w, \alpha)} \textsc{EqTlk}$$

is immediate since we can derive:

$$\frac{\begin{array}{c}\Delta; \cdot; \Sigma_1' \vdash \mathsf{Eq}(A, s, t) \mid e \\ \Delta; \cdot; \cdot \vdash \mathcal{T}^B \mid \theta^w \\ \sigma(\alpha) : \mathsf{Eq}(A) \in \sigma(\Sigma_1)\end{array}}{\Delta; \Gamma; \sigma(\Sigma_1) \vdash \mathsf{Eq}(B, e\theta^w, \sigma(\alpha)\alpha\theta^w) \mid \mathfrak{ti}(\theta^w, \sigma(\alpha))} \textsc{EqTlk}$$

- Suppose the derivation ends in:

$$\frac{\Delta; \Gamma; \Sigma_1 \vdash \mathsf{Eq}(A, s_1, s_2) \mid e}{\Delta; \Gamma; \Sigma_1 \vdash \mathsf{Eq}(A, s_2, s_1) \mid \mathfrak{s}(e)} \mathsf{EqSym}$$

We resort to the IH.

- If the derivation ends in:

$$\frac{\begin{array}{c}\Delta; \Gamma; \Sigma_1 \vdash \mathsf{Eq}(A, s_1, s_2) \mid e_1 \\ \Delta; \Gamma; \Sigma_1 \vdash \mathsf{Eq}(A, s_2, s_3) \mid e_2\end{array}}{\Delta; \Gamma; \Sigma_1 \vdash \mathsf{Eq}(A, s_1, s_3) \mid \mathfrak{t}(e_1, e_2)} \mathsf{EqTrans}$$

We resort to the IH.

- If the derivation ends in:

$$\frac{\Delta; \Gamma, b : A; \Sigma_1 \vdash \mathsf{Eq}(B, s, t) \mid e}{\begin{array}{c}\Delta; \Gamma; \Sigma_1 \vdash \mathsf{Eq}(A \supset B, \lambda b : A.s, \lambda b : A.t) \mid \\ \mathfrak{ab}\mathfrak{C}(b^A.e)\end{array}} \mathsf{EqAbs}$$

We resort to the IH.

- If the derivation ends in:

$$\frac{\begin{array}{c}\Delta; \Gamma_1; \Sigma_{11} \vdash \mathsf{Eq}(A \supset B, s_1, s_2) \mid e_1 \\ \Delta; \Gamma_2; \Sigma_{12} \vdash \mathsf{Eq}(A, t_1, t_2) \mid e_2\end{array}}{\Delta; \Gamma_{1,2}; \Sigma_{11,12} \vdash \mathsf{Eq}(B, s_1 \cdot t_1, s_2 \cdot t_2) \mid \mathfrak{ap}\mathfrak{C}(e_1, e_2)} \mathsf{EqApp}$$

We resort to the IH.

- If the derivation ends in:

$$\frac{\begin{array}{c}\Delta'; \Gamma_1; \Sigma_{11} \vdash \mathsf{Eq}([\![\Sigma.r]\!]A, s_1, s_2) \mid e_1 \\ \Delta', u : A[\Sigma]; \Gamma_2; \Sigma_{12} \vdash \mathsf{Eq}(C, t_1, t_2) \mid e_2\end{array}}{\begin{array}{c}\Delta'; \Gamma_{1,2}; \Sigma_{11,12} \vdash \mathsf{Eq}(C^u_{\Sigma.r}, \text{LET}(u^{A[\Sigma]}.t_1, s_1), \text{LET}(u^{A[\Sigma]}.t_2, s_2)) \mid \\ \mathfrak{le}\mathfrak{C}(u^{A[\Sigma]}.e_2, e_1)\end{array}} \mathsf{EqLet}$$

We resort to the IH and derive:

$$\dfrac{\begin{array}{c}\Delta'; \Gamma_1; \sigma(\Sigma_{11}) \vdash \mathsf{Eq}([\![\Sigma.r]\!]A, \sigma(s_1), \sigma(s_2)) \mid \sigma(e_1) \\ \Delta', u : A[\Sigma]; \Gamma_2; \sigma(\Sigma_{12}) \vdash \mathsf{Eq}(C, \sigma(t_1), \sigma(t_2)) \mid \sigma(e_2)\end{array}}{\begin{array}{c}\Delta'; \Gamma_{1,2}; \sigma(\Sigma_{11,12}) \vdash \mathsf{Eq}(C^u_{\Sigma.r}, \text{LET}(u^{A[\Sigma]}.\sigma(t_1), \sigma(s_1)), \text{LET}(u^{A[\Sigma]}.\sigma(t_2), \sigma(s_2))) \mid \\ \mathfrak{leC}(u^{A[\Sigma]}.\sigma(e_2), \sigma(e_1))\end{array}} \;\mathsf{EqLet}$$

- If the derivation ends in:

$$\dfrac{\Delta; \cdot; \cdot \vdash \mathsf{Eq}(\mathcal{T}^B, \theta'^w, \theta^w) \mid e_i \quad \alpha : \mathsf{Eq}(A) \in \Sigma_1}{\begin{array}{c}\Delta; \Gamma; \Sigma_1 \vdash \mathsf{Eq}(B, \alpha\theta'^w, \alpha\theta^w) \mid \\ \mathfrak{rpC}(\overline{e})\end{array}} \;\textsc{EqRpl}$$

then we derive:

$$\dfrac{\Delta; \cdot; \cdot \vdash \mathsf{Eq}(\mathcal{T}^B, \theta'^w, \theta^w) \mid e_i \quad \sigma(\alpha) : \mathsf{Eq}(A) \in \sigma(\Sigma_1)}{\begin{array}{c}\Delta; \Gamma; \sigma(\Sigma_1) \vdash \mathsf{Eq}(B, \sigma(\alpha)\theta'^w, \sigma(\alpha)\theta^w) \mid \\ \mathfrak{rpC}(\overline{e})\end{array}} \;\textsc{EqRpl}$$

▪

We now address the proof of the Substitution Principle for Validity Hypothesis.

Proof. By simultaneous induction on the derivations of $\Delta_1, u : A[\Sigma_1], \Delta_2; \Gamma; \Sigma_2 \vdash N : C \mid r$ and $\Delta_2, u : A[\Sigma_1], \Delta_2; \Gamma; \Sigma_2 \vdash \mathsf{Eq}(C, s_1, s_2) \mid e$. We write $\Delta^u_{1,2}$ for $\Delta_1, u : A[\Sigma_1], \Delta_2$.

- The derivation ends in:

$$\dfrac{}{\Delta^u_{1,2}; \Gamma; \Sigma_2 \vdash b : A \mid b} \;\mathsf{Var}$$

Then $\Delta_{1,2}; b : A; \Sigma_2 \vdash b : A \mid b$ is derivable.
- If the derivation ends in:

$$\dfrac{\Delta^u_{1,2}; \Gamma, b : A; \Sigma_2 \vdash M : B \mid r_1}{\Delta^u_{1,2}; \Gamma; \Sigma_2 \vdash \lambda b : A.M : A \supset B \mid \lambda b : A.r_1} \;\supset \mathsf{I}$$

then by the IH $\Delta_{1,2}; \Gamma, b : A; \Sigma_2 \vdash M^u_{\Sigma_1.(M,t,e)} : B^u_{\Sigma_1.t} \mid r_1{}^u_{\Sigma_1.t}$. Note that $A^u_{\Sigma_1.t} = A$ since all labels are chosen fresh. We conclude by resorting to $\supset \mathsf{I}$.
- Suppose the derivation ends in:

$$\dfrac{\Delta^u_{1,2}; \Gamma_1; \Sigma_{21} \vdash M_1 : A \supset B \mid s_1 \quad \Delta^u_{1,2}; \Gamma_2; \Sigma_{22} \vdash M_2 : A \mid s_2}{\Delta^u_{1,2}; \Gamma_{1,2}; \Sigma_{21,22} \vdash M_1\, M_2 : B \mid s_1 \cdot s_2} \;\supset \mathsf{E}$$

By the IH we have:
  - $\Delta_{1,2}; \Gamma_1; \Sigma_{21} \vdash M_1{}^u_{\Sigma_1.(M,t,e)} : A^u_{\Sigma_1.t} \supset B^u_{\Sigma_1.t} \mid s_1{}^u_{\Sigma_1.t}$
  - $\Delta^u_{1,2}; \Gamma_2; \Sigma_{22} \vdash M_2{}^u_{\Sigma_1.(M,t,e)} : A^u_{\Sigma_1.t} \mid s_2{}^u_{\Sigma_1.t}$

We conclude by resorting to an instance of $\supset \mathsf{E}$.
- If the derivation ends in:

$$\dfrac{v : C[\Sigma_3] \in \Delta^u_{1,2} \quad \Sigma_3\sigma \subseteq \Sigma_2}{\Delta^u_{1,2}; \Gamma; \Sigma_2 \vdash \langle v; \sigma\rangle : C \mid \langle v; \sigma\rangle} \;\mathsf{mVar}$$

we consider two subcases:
  - $u = v$. First, from Lem. C.2.5 and the hypothesis, we obtain a derivation of $\Delta_{1,2}; \cdot; \Sigma_2 \vdash M\sigma : A \mid s\sigma$ and $\Delta_{1,2}; \cdot; \Sigma_2 \vdash \mathsf{Eq}(A, s\sigma, t\sigma) \mid e\sigma$. We then resort to weakening and construct the derivation:

$$\dfrac{\begin{array}{c}\Delta_{1,2}; \Gamma; \Sigma_2 \vdash M\sigma : A \mid s\sigma \\ \Delta_{1,2}; \Gamma; \Sigma_2 \vdash \mathsf{Eq}(A, s\sigma, t\sigma) \mid e\sigma\end{array}}{\Delta_{1,2}; \Gamma; \Sigma_2 \vdash e\sigma \rhd M\sigma : A \mid t\sigma}$$

Note that $\langle u; \sigma\rangle^u_{\Sigma_1.(M,t,e)} = e\sigma \rhd M\sigma$. Also, $\langle u; \sigma\rangle^u_{\Sigma_1.t} = t\sigma$. Hence we conclude.

– $u \neq v$. Then we derive the following, where Δ'' is $(\Delta_{11}, v : A[\Sigma], \Delta_{12}) \setminus u$:

$$\frac{v : C[\Sigma_3] \in \Delta^u_{1,2} \quad \Sigma_3\sigma \subseteq \Sigma_2}{\Delta_{1,2}; \Gamma; \Sigma_2 \vdash \langle v; \sigma\rangle : A \mid \langle v; \sigma\rangle}\ \mathsf{mVar}$$

- If the derivation ends in:

$$\frac{\Delta^u_{1,2}; \cdot; \Sigma_3 \vdash M_1 : A \mid s_1 \quad \Delta^u_{1,2}; \cdot; \Sigma_3 \vdash \mathsf{Eq}(A, s_1, s_2) \mid e_3}{\Delta^u_{1,2}; \Gamma; \Sigma_2 \vdash !^{\Sigma_3}_{e_3} M_1 : [\![\Sigma_3.s_2]\!]A \mid \Sigma_3.s_2}\ \Box\mathsf{I}$$

By the IH we have:
– $\Delta_{1,2}; \cdot; \Sigma_3 \vdash {M_1}^u_{\Sigma_1.(M,t,e)} : A^u_{\Sigma_1.t} \mid {s_1}^u_{\Sigma_1.t}$
– $\Delta_{1,2}; \cdot; \Sigma_3 \vdash \mathsf{Eq}(A^u_{\Sigma_1.t}, {s_1}^u_{\Sigma_1.t}, {s_2}^u_{\Sigma_1.t}) \mid {e_3}^u_{\Sigma_1.t}$
We then construct the derivation

$$\frac{\begin{array}{c}\Delta_{1,2}; \cdot; \Sigma_3 \vdash {M_1}^u_{\Sigma_1.(M,t,e)} : A^u_{\Sigma_1.t} \mid {s_1}^u_{\Sigma_1.t} \\ \Delta_{1,2}; \cdot; \Sigma_3 \vdash \mathsf{Eq}(A^u_{\Sigma_1.t}, {s_1}^u_{\Sigma_1.t}, {s_2}^u_{\Sigma_1.t}) \mid {e_3}^u_{\Sigma_1.t}\end{array}}{\Delta_{1,2}; \Gamma; \Sigma_2 \vdash !^{\Sigma_3}_{{e_3}^u_{\Sigma_1.t}} {M_1}^u_{\Sigma_1.(M,t,e)} : [\![\Sigma_3.{s_2}^u_{\Sigma_1.t}]\!]A \mid \Sigma_3.{s_2}^u_{\Sigma_1.t}}\ \Box\mathsf{I}$$

- Suppose $q \overset{\text{def}}{=} \textsc{let}(u^{A[\Sigma]}.s_2, s_1)$ and the derivation ends in:

$$\frac{\begin{array}{c}\Delta^u_{1,2}; \Gamma_1; \Sigma_{21} \vdash M_1 : [\![\Sigma.r]\!]A \mid s_1 \\ \Delta^u_{1,2}, v : A[\Sigma]; \Gamma_2; \Sigma_{22} \vdash M_2 : C \mid s_2\end{array}}{\Delta^u_{1,2}; \Gamma_{1,2}; \Sigma_{21,22} \vdash \mathsf{let}\, u : A[\Sigma] = M_1 \;\mathsf{in}\; M_2 : C^u_{\Sigma.r} \mid q}\ \Box\mathsf{E}$$

We resort to the IH and derive the following derivation where $P \overset{\text{def}}{=} \mathsf{let}\, v : A[\Sigma] = {M_1}^u_{\Sigma_1.(M,t,e)} \;\mathsf{in}\; {M_2}^u_{\Sigma_1.t}$ and $q' \overset{\text{def}}{=} \textsc{let}(u^{A[\Sigma]}.{s_2}^u_{\Sigma_1.t}, {s_1}^u_{\Sigma_1.t})$:

$$\frac{\begin{array}{c}\Delta_{1,2}; \Gamma_1; \Sigma_{21} \vdash {M_1}^u_{\Sigma_1.(M,t,e)} : [\![\Sigma.r^u_{\Sigma_1.t}]\!]A^u_{\Sigma_1.t} \mid {s_1}^u_{\Sigma_1.t} \\ \Delta_{1,2}, v : A[\Sigma]; \Gamma_2; \Sigma_{22} \vdash {M_2}^u_{\Sigma_1.(M,t,e)} : C^u_{\Sigma_1.t} \mid {s_2}^u_{\Sigma_1.t}\end{array}}{\Delta_{1,2}; \Gamma_{1,2}; \Sigma_{21,22} \vdash P : {C^u_{\Sigma_1.t}}^u_{\Sigma.r^u_{\Sigma_1.t}} \mid q'}\ \Box\mathsf{E}$$

- If the derivation ends in:

$$\frac{\alpha : \mathsf{Eq}(A) \in \Sigma_2 \quad \Delta^u_{1,2}; \cdot; \cdot \vdash \theta : \mathcal{T}^B \mid \theta^w}{\Delta^u_{1,2}; \Gamma; \Sigma_2 \vdash \alpha\theta : B \mid \alpha\theta^w}\ \text{TT}\textsc{lk}$$

We resort to the IH and derive:

$$\frac{\begin{array}{c}\alpha : \mathsf{Eq}(A) \in \Sigma_2 \\ \Delta_{1,2}; \cdot; \cdot \vdash \theta^u_{\Sigma_1.(M,t,e)} : \mathcal{T}^{B^u_{\Sigma_1.t}} \mid {\theta^w}^u_{\Sigma_1.t}\end{array}}{\Delta_{1,2}; \Gamma; \Sigma_2 \vdash \alpha(\theta^u_{\Sigma_1.(M,t,e)}) : B^u_{\Sigma_1.t} \mid \alpha({\theta^w}^u_{\Sigma_1.t})}\ \text{TT}\textsc{lk}$$

- If the derivation ends in:

$$\frac{\Delta^u_{1,2}; \Gamma; \Sigma_2 \vdash M_1 : A \mid s_1}{\Delta^u_{1,2}; \Gamma; \Sigma_2 \vdash \mathsf{Eq}(A, s, s) \mid \mathfrak{r}(s)}\ \mathsf{EqRefl}$$

We use the IH and derive:

$$\frac{\Delta_{1,2}; \Gamma; \Sigma_2 \vdash {M_1}^u_{\Sigma_1.(M,t,e)} : A^u_{\Sigma_1.t} \mid {s_1}^u_{\Sigma_1.t}}{\Delta_{1,2}; \Gamma; \Sigma_2 \vdash \mathsf{Eq}(A^u_{\Sigma_1.t}, {s_1}^u_{\Sigma_1.t}, {s_1}^u_{\Sigma_1.t}) \mid \mathfrak{r}({s_1}^u_{\Sigma_1.t})}\ \mathsf{EqRefl}$$

- If the derivation ends in:

$$\frac{\begin{array}{c}\Delta^u_{1,2}; \Gamma; \Sigma_2 \vdash M_3 : A \mid s_1 \\ \Delta^u_{1,2}; \Gamma; \Sigma_2 \vdash \mathsf{Eq}(A, s_1, s_2) \mid e_3\end{array}}{\Delta^u_{1,2}; \Gamma; \Sigma_2 \vdash e_3 \rhd M_3 : A \mid s_2}\ \mathsf{Eq}$$

We use the IH and derive:

$$\frac{\begin{array}{c}\Delta_{1,2};\Gamma;\Sigma_2 \vdash {M_3}^{u}_{\Sigma_1.(M,t,e)} : A^{u}_{\Sigma_1.t} \mid {s_1}^{u}_{\Sigma_1.t}\\ \Delta_{1,2};\Gamma;\Sigma_2 \vdash \mathsf{Eq}(A^{u}_{\Sigma_1.t}, {s_1}^{u}_{\Sigma_1.t}, {s_2}^{u}_{\Sigma_1.t}) \mid {e_3}^{u}_{\Sigma_1.t}\end{array}}{\Delta_{1,2};\Gamma;\Sigma_2 \vdash {e_3}^{u}_{\Sigma_1.t} \rhd {M_3}^{u}_{\Sigma_1.t} : A^{u}_{\Sigma_1.t} \mid {s_2}^{u}_{\Sigma_1.t}}\ \mathsf{Eq}$$

- If the derivation ends in:

$$\frac{\begin{array}{c}\Delta^{u}_{1,2};\Gamma_1, b:A;\Sigma_{21} \vdash M_1 : B \mid r_1\\ \Delta^{u}_{1,2};\Gamma_2;\Sigma_{22} \vdash M_2 : A \mid r_2\end{array}}{\Delta^{u}_{1,2};\Gamma_{1,2};\Sigma_{21,22} \vdash \mathsf{Eq}(B, r^{b}_{r_2}, (\lambda b:A.r_1)\cdot r_2) \mid \mathfrak{ba}(b^A.r_1, r_2)}\ \mathsf{Eq}\beta$$

  We use the IH and derive:

$$\frac{\begin{array}{c}\Delta_{1,2};\Gamma_1, b:A;\Sigma_{21} \vdash {r_1}^{u}_{\Sigma_1.(M,t,e)} : B \mid r\\ \Delta_{1,2};\Gamma_2;\Sigma_{22} \vdash {M_2}^{u}_{\Sigma_1.(M,t,e)} : A^{u}_{\Sigma_1.t} \mid {r_2}^{u}_{\Sigma_1.t}\end{array}}{\Delta_{1,2};\Gamma_{1,2};\Sigma_{21,22} \vdash \mathsf{Eq}(B, {{r_1}^{u}_{\Sigma_1.t}}^{b}_{{r_2}^{u}_{\Sigma_1.t}}, (\lambda b:A.{r_1}^{u}_{\Sigma_1.t})\cdot {r_2}^{u}_{\Sigma_1.t}) \mid \mathfrak{ba}(b^A.{r_1}^{u}_{\Sigma_1.t}, {r_2}^{u}_{\Sigma_1.t})}\ \mathsf{Eq}\beta$$

- Let $q \overset{\text{def}}{=} \text{LET}(v^{A[\Sigma]}.t_1, \Sigma.s)$ and $e_2 \overset{\text{def}}{=} \mathfrak{bb}(v^{A[\Sigma]}.t_1, \Sigma.s)$. If the derivation ends in:

$$\frac{\begin{array}{c}\Delta^{u}_{1,2};\cdot;\Sigma \vdash A \mid r\\ \Delta^{u}_{1,2};\cdot;\Sigma \vdash \mathsf{Eq}(A, r, s) \mid e\\ \Delta^{u}_{1,2}, v:A[\Sigma];\Gamma;\Sigma_2 \vdash C \mid t_1\end{array}}{\Delta^{u}_{1,2};\Gamma;\Sigma_2 \vdash \mathsf{Eq}(C^{v}_{\Sigma.s}, {t_1}^{v}_{\Sigma.s}, q) \mid e_2}\ \mathsf{Eq}\beta_{\Box}$$

  We use the IH and derive the following derivation where $q' \overset{\text{def}}{=} \text{LET}(v^{A[\Sigma]}.{t_1}^{u}_{\Sigma_1.t}, \Sigma.s^{u}_{\Sigma_1.t})$ and $e'_2 \overset{\text{def}}{=} \mathfrak{bb}(v^{A[\Sigma]}.{t_1}^{u}_{\Sigma_1.t}, \Sigma.s^{u}_{\Sigma_1.t})$:

$$\frac{\begin{array}{c}\Delta_{1,2};\cdot;\Sigma \vdash A^{u}_{\Sigma_1.t} \mid r^{u}_{\Sigma_1.t}\\ \Delta_{1,2};\cdot;\Sigma \vdash \mathsf{Eq}(A^{u}_{\Sigma_1.t}, r^{u}_{\Sigma_1.t}, s^{u}_{\Sigma_1.t}) \mid e^{u}_{\Sigma_1.t}\\ \Delta_{1,2}, v:A[\Sigma];\Gamma;\Sigma_2 \vdash C^{u}_{\Sigma_1.t} \mid {t_1}^{u}_{\Sigma_1.t}\end{array}}{\Delta_{1,2};\Gamma;\Sigma_2 \vdash \mathsf{Eq}({C^{u}_{\Sigma_1.t}}^{v}_{\Sigma.s}, ({t_1}^{u}_{\Sigma_1.t})^{v}_{\Sigma.s^{u}_{\Sigma_1.t}}, q') \mid e'_2}\ \mathsf{Eq}\beta_{\Box}$$

  Note that, by the freshness condition on labels of hypothesis, $u \notin A$. Therefore, the first hypothesis reads: $\Delta_{1,2};\cdot;\Sigma \vdash A \mid r^{u}_{\Sigma_1.t}$ and the second one reads $\Delta_{1,2};\cdot;\Sigma \vdash \mathsf{Eq}(A, r^{u}_{\Sigma_1.t}, s^{u}_{\Sigma_1.t}) \mid e^{u}_{\Sigma_1.t}$, hence the above derivation is a valid one. We conclude by resorting to Lem. C.2.4(3).
- The case in which the derivation ends in:

$$\frac{\begin{array}{c}\Delta^{u}_{1,2};\cdot;\Sigma_1 \vdash \mathsf{Eq}(A, s, t) \mid e\\ \Delta^{u}_{1,2};\cdot;\cdot \vdash \mathcal{T}^B \mid \theta^w\\ \alpha : \mathsf{Eq}(A) \in \Sigma_2\end{array}}{\Delta^{u}_{1,2};\Gamma;\Sigma_2 \vdash \mathsf{Eq}(B, e\theta^w, \alpha\theta^w) \mid \mathfrak{ti}(\theta^w, \alpha)}\ \text{EqTlk}$$

  We use the IH.
- Suppose the derivation ends EqSym or $\mathsf{EqTrans}$, then we proceed similarly to the case of EqRefl.
- If the derivation ends in:

$$\frac{\Delta^{u}_{1,2};\Gamma, b:A;\Sigma_2 \vdash \mathsf{Eq}(B, s, t) \mid e}{\Delta^{u}_{1,2};\Gamma;\Sigma_2 \vdash \mathsf{Eq}(A \supset B, \lambda b:A.s, \lambda b:A.t) \mid \mathfrak{abC}(b^A.e)}\ \mathsf{EqAbs}$$

  We use the IH.
- If the derivation ends in:

$$\frac{\begin{array}{c}\Delta^{u}_{1,2};\Gamma_1;\Sigma_{21} \vdash \mathsf{Eq}(A \supset B, s_1, s_2) \mid e_1\\ \Delta^{u}_{1,2};\Gamma_2;\Sigma_{22} \vdash \mathsf{Eq}(A, t_1, t_2) \mid e_2\end{array}}{\Delta^{u}_{1,2};\Gamma_{1,2};\Sigma_{21,22} \vdash \mathsf{Eq}(B, s_1\cdot t_1, s_2\cdot t_2) \mid \mathfrak{apC}(e_1, e_2)}\ \mathsf{EqApp}$$

We use the IH.

- Let $q_1 \overset{\text{def}}{=} \textsc{let}(v^{A[\Sigma]}.t_1, s_1)$ and $q_2 \overset{\text{def}}{=} \textsc{let}(v^{A[\Sigma]}.t_2, s_2)$. If the derivation ends in:

$$\frac{\begin{array}{c}\Delta^u_{1,2}; \Gamma_1; \Sigma_{21} \vdash \mathsf{Eq}([\![\Sigma.r]\!]A, s_1, s_2) \mid e_3 \\ \Delta^u_{1,2}, v : A[\Sigma]; \Gamma_2; \Sigma_{22} \vdash \mathsf{Eq}(C, t_1, t_2) \mid e_4\end{array}}{\Delta^u_{1,2}; \Gamma_{1,2}; \Sigma_{21,22} \vdash \mathsf{Eq}(C^v_{\Sigma.r}, q_1, q_2) \mid \mathfrak{leC}(v^{A[\Sigma]}.e_4, e_3)}\ \mathsf{EqLet}$$

We use the IH and Lem. C.2.4(3).

- If the derivation ends in:

$$\frac{\Delta^u_{1,2}; \cdot; \cdot \vdash \mathsf{Eq}(\mathcal{T}^B, \theta^w, \theta'^w) \mid e_i \quad \alpha : \mathsf{Eq}(A) \in \Sigma_2}{\Delta^u_{1,2}; \Gamma; \Sigma_2 \vdash \mathsf{Eq}(B, \alpha\theta'^w, \alpha\theta^w) \mid \mathfrak{rpC}(\overline{e})}\ \textsc{EqRpl}$$

We use the IH and derive:

$$\frac{\begin{array}{c}\Delta_{1,2}; \cdot; \cdot \vdash \mathsf{Eq}(\mathcal{T}^{B^u_{\Sigma_1.t}}, \theta'{w}^u_{\Sigma_1.t}, \theta{w}^u_{\Sigma_1.t}) \mid \overline{e}^u_{\Sigma_1.t} \\ \alpha : \mathsf{Eq}(A) \in \Sigma_2\end{array}}{\Delta_{1,2}; \Gamma; \Sigma_2 \vdash \mathsf{Eq}(B^u_{\Sigma_1.t}, \alpha(\theta'{w}^u_{\Sigma_1.t}), \alpha(\theta{w}^u_{\Sigma_1.t})) \mid \mathfrak{rpC}(\overline{e}^u_{\Sigma_1.t})}\ \textsc{EqRpl}$$

■

C.3 Subject Reduction

Proof of Prop. 5.6.4.

Proof. By induction on the derivation of $\cdot; \cdot; \Sigma \vdash M : A \mid s$. Note that if the derivation ends in TVar or TMVar, the result holds trivially. Also, if it ends in TAbs, then M is an abstraction and hence a value. Finally, if the derivation ends in an instance of TEq, the result holds trivially too since M is assumed in permutation reduction-normal form. We consider the remaining cases:

- Suppose $M = M_1\, M_2$ and the derivation ends in an instance of TApp. Then M_1 (and M_2) is also in permutation reduction normal form, typable and tv-closed. We apply the IH on M_1 and reason as follows:
  1. M_1 is a value V. By Canonical Forms Lem. 5.6.3 V is an abstraction $\lambda a^B.M_3$. We resort to the IH on M_2:
     a) M_2 is a value. Then $(\lambda a^B.M_3)\, M_2 \mapsto \mathfrak{ba}(a^B.s, t) \rhd {M_3}^a_{M_2,t}$.
     b) M_2 is inspection-blocked. Then $V\, M_2$ is inspection-blocked too.
     c) There exists M_2' s.t. $M_2 \mapsto M_2'$. Then $V\, M_2 \mapsto V\, M_2'$.
  2. M_1 is inspection-blocked. Then $M_1\, M_2$ is inspection-blocked.
  3. There exists M_1' s.t. $M_1 \mapsto M_1'$. Then $M_1\, M_2 \mapsto M_1'\, M_2$.
- $M = !^{\Sigma_1}_e M_1$ and the derivation ends in TBox. Since M_1 is in permutation reduction normal form, typed and tv-closed, we resort to the IH and proceed as follows:
  1. M_1 is a value V. In this case $!^{\Sigma_1}_e V$ is also a value.
  2. M_1 is inspection-blocked. Then, for some $\mathcal{F}$, α, θ, $M_1 = \mathcal{F}[\alpha\theta]$. Then $!^{\Sigma_1}_e \mathcal{F}[\alpha\vartheta] \mapsto !^{\Sigma_1}_e \mathcal{F}[\mathfrak{ti}(\theta, \alpha) \rhd e\vartheta]$.
  3. There exists M_1' s.t. $M_1 \mapsto M_1'$. Then $!^{\Sigma_1}_e M_1 \mapsto !^{\Sigma_1}_e M_1'$.
- $M = \textsc{let}(u^{A[\Sigma]}.M_2, M_1)$ and the derivation ends in an instance of TLetB. Since M_1 is in permutation reduction normal form, typable and tv-closed we can apply the IH and reason as follows:
  1. M_1 is a value. By the Canonical Forms Lem. 5.6.3, M_1 is an audited computation unit $!^{\Sigma}_e V$. Then $M = \textsc{let}(u^{A[\Sigma]}.M_2, !^{\Sigma}_e V) \mapsto \mathfrak{bb}(u^{A[\Sigma]}.t, \Sigma.s) \rhd {M_2}^u_{\Sigma.(V,s,e)}$.
  2. M_1 is inspection-blocked. Then, for some $\mathcal{F}$, α, ϑ, $M_1 = \mathcal{F}[\alpha\vartheta]$. Therefore, $\textsc{let}(u^{A[\Sigma]}.M_3, \mathcal{F}[\alpha\vartheta])$ is inspection-blocked.
  3. There exists M_1' s.t. $M_1 \mapsto M_1'$. Then $\textsc{let}(u^{A[\Sigma]}.M_2, M_1) \mapsto \textsc{let}(u^{A[\Sigma]}.M_2, M_1')$.

- $M = \alpha\theta$ where $\theta = \{c_1/M_1, \ldots, c_{10}/M_{10}\}$. Since each M_i, $i \in 1..10$, is in permutation reduction normal form, typable and tv-closed we may apply the IH to each one. If they are all values, then M is inspection-blocked. Otherwise, let M_j be the first that is not a value. Then, it must be the case that $M_j \mapsto M'_j$ for some M'_j. Then also, $M \mapsto \alpha\theta'$, where $\theta'(c_i) \overset{\text{def}}{=} M_i$, $i \in 1..10/j$ and $\theta'(c_j) \overset{\text{def}}{=} M'_j$.

▪

C.4 Strong Normalisation

Proof of Lem. 5.7.1 ($\mathcal{S}()$ preserves typability).

Proof. By induction on the derivation of $\Delta; \Gamma; \Sigma \vdash M : A \mid s$. The interesting cases are:

- The derivation ends in:

$$\frac{\Delta; \cdot; \Sigma \vdash M : A \mid s \quad \Delta; \cdot; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e}{\Delta; \Gamma; \Sigma' \vdash !^{\Sigma}_{e} M : [\![\Sigma.t]\!]A \mid \Sigma.t} \text{TBox}$$

  By the IH $\mathcal{S}(\Delta) \vdash \mathcal{S}(M) : \mathcal{S}(A)$ is derivable. We resort to weakening and conclude.
- $M = \text{LET}\, u^{A[\Sigma]} \leftarrow M_1 \,\text{IN}\, M_2$ and the derivation ends in:

$$\frac{\begin{array}{c}\Delta; \Gamma_1; \Sigma_1 \vdash M_1 : [\![\Sigma.r]\!]A \mid s \\ \Delta, u : A[\Sigma]; \Gamma_2; \Sigma_2 \vdash M_2 : C \mid t\end{array}}{\Delta; \Gamma_{1,2}; \Sigma_{1,2} \vdash M : C^{u}_{\Sigma.r} \mid \text{LET}(u^{A[\Sigma]}.t, s)} \text{TLetB}$$

  By the IH both $\mathcal{S}(\Delta), \mathcal{S}(\Gamma_1) \vdash \mathcal{S}(M_1) : \mathcal{S}(A)$ and $\mathcal{S}(\Delta), u : \mathcal{S}(A), \mathcal{S}(\Gamma_2) \vdash \mathcal{S}(M_2) : \mathcal{S}(C)$ are derivable. We use weakening and then construct the derivation:

$$\frac{\dfrac{\mathcal{S}(\Delta), u : \mathcal{S}(A), \mathcal{S}(\Gamma_{1,2}) \vdash \mathcal{S}(M_2) : \mathcal{S}(C)}{\mathcal{S}(\Delta), \mathcal{S}(\Gamma_{1,2}) \vdash \lambda u : \mathcal{S}(A).\mathcal{S}(M_2) : \mathcal{S}(A) \supset \mathcal{S}(C)} \quad \mathcal{S}(\Delta), \mathcal{S}(\Gamma_{1,2}) \vdash \mathcal{S}(M_1) : \mathcal{S}(A)}{\mathcal{S}(\Delta), \mathcal{S}(\Gamma_{1,2}) \vdash (\lambda u : \mathcal{S}(A).\mathcal{S}(M_2))\, \mathcal{S}(M_1) : \mathcal{S}(C)}$$

  Finally, note that $\mathcal{S}(C^{u}_{\Sigma.s}) = \mathcal{S}(C)$.
- The derivation ends in:

$$\frac{\alpha : \mathsf{Eq}(A) \in \Sigma \quad \Delta; \cdot; \cdot \vdash \vartheta : \mathcal{T}^{B} \mid \theta}{\Delta; \Gamma; \Sigma \vdash \alpha^{B}\vartheta : B \mid \alpha\theta} \text{TTlk}$$

  By the IH, for each compatibility witness constructor c, $\mathcal{S}(\Delta) \vdash \mathcal{S}(\vartheta(c)) : \mathcal{S}(\mathcal{T}^{B}(c))$ is derivable. We resort to weakening to obtain derivations of $\mathcal{S}(\Delta), \mathcal{S}(\Gamma) \vdash \mathcal{S}(\vartheta(c)) : \mathcal{S}(\mathcal{T}^{B}(c))$. With this we can produce the derivation π:

$$\frac{\mathcal{S}(\Delta), \mathcal{S}(\Gamma) \vdash \mathcal{S}(\vartheta) : \mathcal{S}(\mathcal{T}^{B})}{\mathcal{S}(\Delta), \mathcal{S}(\Gamma) \vdash \mathcal{S}(\vartheta) : \mathcal{S}(\mathcal{T}^{B})} \text{Prod}$$

  Now we construct:

$$\frac{\mathcal{S}(\Delta), \mathcal{S}(\Gamma) \vdash \lambda a : \mathcal{S}(\mathcal{T}^{B}).c_B : \mathcal{S}(\mathcal{T}^{B}) \supset \mathcal{S}(B) \quad \pi}{\mathcal{S}(\Delta), \mathcal{S}(\Gamma) \vdash (\lambda a : \mathcal{S}(\mathcal{T}^{B}).c_B)\, \mathcal{S}(\vartheta) : \mathcal{S}(B)} \text{TApp}$$

- The derivation ends in:

$$\frac{\Delta; \Gamma; \Sigma \vdash M : A \mid s \quad \Delta; \Gamma; \Sigma \vdash \mathsf{Eq}(A, s, t) \mid e}{\Delta; \Gamma; \Sigma \vdash e \rhd M : A \mid t} \text{TEq}$$

  We conclude by the IH from which we deduce the derivability of $\mathcal{S}(\Delta), \mathcal{S}(\Gamma) \vdash \mathcal{S}(M) : \mathcal{S}(A)$.

▪

Proof of Lem. 5.7.4

Proof. Suppose $M = \mathcal{D}[M']$ and $N = \mathcal{D}[N']$ with $M' \xrightarrow{f} N'$. We proceed by induction on $\mathcal{D}$.

- $\mathcal{D} = \Box$. We consider each of the two principle reduction axioms β and $\beta_\Box$:
  - β. We reason as follows:

$$\begin{array}{l} \mathcal{S}((\lambda a : A.M)\ N) \\ = (\lambda a : \mathcal{S}(A).\mathcal{S}(M))\, \mathcal{S}(N) \\ \xrightarrow{\beta} \mathcal{S}(M)^a_{\mathcal{S}(N)} \\ = \mathcal{S}(M^a_{N,t}) \\ = \mathcal{S}(\mathfrak{ba}(a^A.s, t) \rhd M^a_{N,t}) \end{array}$$

  - $\beta_\Box$. We reason as follows:

$$\begin{array}{l} \mathcal{S}(\text{LET}\, u^{A[\Sigma]} \leftarrow !^{\Sigma}_{e} M \,\text{IN}\, N) \\ = (\lambda u : \mathcal{S}(A).\mathcal{S}(N))\, \mathcal{S}(M) \\ \xrightarrow{\beta} \mathcal{S}(N)^u_{\mathcal{S}(M)} \\ = \mathcal{S}(N^u_{\Sigma.(M,t,e)}) \\ = \mathcal{S}(\mathfrak{bb}(u^{A[\Sigma]}.s, \Sigma.t) \rhd N^u_{\Sigma.(M,t,e)}) \end{array}$$

- $\mathcal{D} = \mathcal{D}'\ M_1$. By the IH $\mathcal{S}(\mathcal{D}'[M']) \xrightarrow{\beta} \mathcal{S}(\mathcal{D}'[N'])$. Therefore, also $\mathcal{S}(\mathcal{D}'[M'])\, \mathcal{S}(M_1) \xrightarrow{\beta} \mathcal{S}(\mathcal{D}'[N'])\, \mathcal{S}(M_1)$.
- $\mathcal{D} = M_1\ \mathcal{D}'$. Similar to the previous case.
- $\mathcal{D} = \text{LET}\, u^{A[\Sigma]} \leftarrow \mathcal{D}' \,\text{IN}\, M_1$. By the IH $\mathcal{S}(\mathcal{D}'[M']) \xrightarrow{\beta} \mathcal{S}(\mathcal{D}'[N'])$. Therefore, also $(\lambda u : \mathcal{S}(A).\mathcal{S}(M_1))\, \mathcal{S}(\mathcal{D}'[M']) \xrightarrow{\beta} (\lambda u : \mathcal{S}(A).\mathcal{S}(M_1))\, \mathcal{S}(\mathcal{D}'[N'])$.
- $\mathcal{D} = \text{LET}\, u^{A[\Sigma]} \leftarrow M_1 \,\text{IN}\, \mathcal{D}'$. Similar to the previous item.
- $\mathcal{D} = !^{\Sigma}_{e}\mathcal{D}'$. Immediate from the IH.
- $\mathcal{D} = e \rhd \mathcal{D}'$. Immediate from the IH.
- $\mathcal{D} = \alpha\{c_1/M_1, \ldots, c_j/M_j, c_{j+1}/\mathcal{D}', \ldots\}$. By the IH, $\mathcal{S}(\mathcal{D}'[M']) \xrightarrow{\beta} \mathcal{S}(\mathcal{D}'[N'])$. Therefore, also $(\lambda a : \mathcal{S}(\mathcal{T}^B).c_B)\, \langle \mathcal{S}(M_1), \ldots, \mathcal{S}(M_j), \mathcal{S}(\mathcal{D}'[M']), \ldots, \rangle \xrightarrow{\beta} (\lambda a : \mathcal{S}(\mathcal{T}^B).c_B)\, \langle \mathcal{S}(M_1), \ldots, \mathcal{S}(M_j), \mathcal{S}(\mathcal{D}'[N']), \ldots, \rangle$.

▪

C.4.1 Weight Functions and Associated Results

Note that if there are no audited computation units, then the weight is the empty multiset. The proof of this observation is by close inspection of the definition of the weight functions.

Lemma C.4.1 If M has no occurrences of the trail computation constructor "!", then $\mathcal{W}_n(M) = \langle\!\langle\ \rangle\!\rangle$.

The weight functions are weakly monotonic in their parameter.

Lemma C.4.2 1. $m \leq n$ implies $\mathcal{W}_m(M) \preceq \mathcal{W}_n(M)$.
2. $m < n$ and M has at least one occurrence of "!" implies $\mathcal{W}_m(M) \prec \mathcal{W}_n(M)$.

Proof. By induction on M.

- $M = a$. Then $\mathcal{W}_m(M) = \langle\!\langle\ \rangle\!\rangle = \mathcal{W}_n(M)$.
- $M = \lambda a : A.M_1$.

$$\begin{array}{ll} \mathcal{W}_m(\lambda a : A.M_1) & \\ = \mathcal{W}_m(M_1) & \\ \preceq \mathcal{W}_n(M_1) & \text{IH} \\ = \mathcal{W}_n(\lambda a : A.M_1) & \end{array}$$

- $M = M_1\, M_2$.

$$\begin{aligned}&\mathcal{W}_m(M_1\, M_2)\\ &= \mathcal{W}_m(M_1) \uplus \mathcal{W}_m(M_2)\\ &\preceq \mathcal{W}_n(M_1) \uplus \mathcal{W}_n(M_2) \quad \text{IH} * 2\\ &= \mathcal{W}_n(M_1\, M_2)\end{aligned}$$

- $M = \langle u; \sigma\rangle$. Then $\mathcal{W}_m(M) = \langle\!\langle\ \rangle\!\rangle = \mathcal{W}_n(M)$.
- $M = !^{\Sigma}_{e} M_1$.

$$\begin{aligned}&\mathcal{W}_m(!^{\Sigma}_{e} M_1)\\ &= m * \mathcal{W}^t(M_1) \uplus \mathcal{W}_{m*\mathcal{W}^t(M_1)}(M_1)\\ &\preceq n * \mathcal{W}^t(M_1) \uplus \mathcal{W}_{m*\mathcal{W}^t(M_1)}(M_1)\\ &\preceq n * \mathcal{W}^t(M_1) \uplus \mathcal{W}_{n*\mathcal{W}^t(M_1)}(M_1) \quad \text{IH}\\ &= \mathcal{W}_n(!^{\Sigma}_{e} M_1)\end{aligned}$$

- $M = \mathsf{let}\, u : A[\Sigma] = M_1 \;\mathsf{in}\; M_2$.

$$\begin{aligned}&\mathcal{W}_m(\mathsf{let}\, u : A[\Sigma] = M_1 \;\mathsf{in}\; M_2)\\ &= \mathcal{W}_m(M_1) \uplus \mathcal{W}_m(M_2)\\ &\preceq \mathcal{W}_n(M_1) \uplus \mathcal{W}_n(M_1) \qquad \text{IH} * 2\\ &= \mathcal{W}_n(\mathsf{let}\, u : A[\Sigma] = M_1 \;\mathsf{in}\; M_2)\end{aligned}$$

- $M = \alpha\theta$.

$$\begin{aligned}&\mathcal{W}_m(\alpha\theta)\\ &= \biguplus_{i\in 1..10} \mathcal{W}_m(\theta(c_i))\\ &\preceq \biguplus_{i\in 1..10} \mathcal{W}_n(\theta(c_i)) \quad \text{IH} * 10\\ &= \mathcal{W}_n(\alpha\theta)\end{aligned}$$

- $M = e \rhd M_1$.

$$\begin{aligned}&\mathcal{W}_m(e \rhd M_1)\\ &= \mathcal{W}_m(M_1)\\ &\preceq \mathcal{W}_n(M_1) \qquad \text{IH}\\ &= \mathcal{W}_n(e \rhd M_1)\end{aligned}$$

▪

Lemma C.4.3 $\mathcal{W}_n(M^{a}_{N,t}) \preceq \mathcal{W}_n(M) \uplus \mathcal{W}_n(N)$.

Proof. By induction on M.

- $M = b$. If $b \neq a$, then $\langle\!\langle\ \rangle\!\rangle \preceq \mathcal{W}_n(b) \uplus \mathcal{W}_n(N)$. If $b = a$, then $\mathcal{W}_n(N) \preceq \mathcal{W}_n(N) = \mathcal{W}_n(a) \uplus \mathcal{W}_n(N)$.
- $M = \lambda b : A.M_1$. We reason as follows

$$\begin{aligned}&\mathcal{W}_n((\lambda b : A.M_1)^{a}_{N,t})\\ &= \mathcal{W}_n({M_1}^{a}_{N,t})\\ &\preceq \mathcal{W}_n(M_1) \uplus \mathcal{W}_n(N) \qquad \text{IH}\\ &= \mathcal{W}_n(\lambda b : A.M_1) \uplus \mathcal{W}_n(N)\end{aligned}$$

- $M = M_1\, M_2$. This case relies on the affine nature of truth variables. Note that $\mathsf{fvT}(M_1) \cap \mathsf{fvT}(M_2) = \emptyset$. Suppose, therefore, that $a \in \mathsf{fvT}(M_1)$ (the case $a \in \mathsf{fvT}(M_2)$ is similar and hence omitted). We reason as follows:

$$\begin{aligned}&\mathcal{W}_n((M_1\, M_2)^{a}_{N,t})\\ &= \mathcal{W}_n({M_1}^{a}_{N,t}\, M_2)\\ &= \mathcal{W}_n({M_1}^{a}_{N,t}) \uplus \mathcal{W}_n(M_2)\\ &\preceq (\mathcal{W}_n(M_1) \uplus \mathcal{W}_n(N)) \uplus \mathcal{W}_n(M_2) \quad \text{IH}\\ &= \mathcal{W}_n(M_1\, M_2) \uplus \mathcal{W}_n(N) \qquad \text{Ass. } \uplus\end{aligned}$$

- $M = \langle u; \sigma \rangle$.

$$\begin{array}{l} \mathcal{W}_n(\langle u;\sigma\rangle^a_{N,t}) \\ = \mathcal{W}_n(\langle u;\sigma\rangle) \\ = \langle\!\langle\ \rangle\!\rangle \\ \preceq \mathcal{W}_n(N) \\ = \langle\!\langle\ \rangle\!\rangle \uplus \mathcal{W}_n(N) \\ = \mathcal{W}_n(\langle u;\sigma\rangle) \uplus \mathcal{W}_n(N) \end{array}$$

- $M = !^{\Sigma}_e M_1$.

$$\begin{array}{l} \mathcal{W}_n((!^{\Sigma}_e M_1)^a_{N,t}) \\ = \mathcal{W}_n(!^{\Sigma}_e M_1) \\ \preceq \mathcal{W}_n(!^{\Sigma}_e M_1) \uplus \mathcal{W}_n(N) \end{array}$$

- $M = \mathsf{let}\, u : A[\Sigma] = M_1 \,\mathsf{in}\, M_2$. Note that $\mathsf{fvT}(M_1) \cap \mathsf{fvT}(M_2) = \emptyset$. Suppose, therefore, that $a \in \mathsf{fvT}(M_1)$ (the case $a \in \mathsf{fvT}(M_2)$ is similar and hence omitted). We reason as follows:

$$\begin{array}{ll} \mathcal{W}_n((\mathsf{let}\, u : A[\Sigma] = M_1 \,\mathsf{in}\, M_2)^a_{N,t}) & \\ = \mathcal{W}_n({M_1}^a_{N,t}) \uplus \mathcal{W}_n(M_2) & \\ \preceq (\mathcal{W}_n(M_1) \uplus \mathcal{W}_n(N)) \uplus \mathcal{W}_n(M_2) & \text{IH} \\ = \mathcal{W}_n(\mathsf{let}\, u : A[\Sigma] = M_1 \,\mathsf{in}\, M_2) \uplus \mathcal{W}_n(N) & \text{Ass. } \uplus \end{array}$$

- $M = \alpha\theta$.

$$\begin{array}{l} \mathcal{W}_n((\alpha\theta)^a_{N,t}) \\ = \mathcal{W}_n(\alpha\theta) \\ \preceq \mathcal{W}_n(\alpha\theta) \uplus \mathcal{W}_n(N) \end{array}$$

- $M = e \rhd M_1$.

$$\begin{array}{ll} \mathcal{W}_n((e \rhd M_1)^a_{N,t}) & \\ = \mathcal{W}_n(e^a_t \rhd {M_1}^a_{N,t}) & \\ = \mathcal{W}_n({M_1}^a_{N,t}) & \\ \preceq \mathcal{W}_n(M_1) \uplus \mathcal{W}_n(N) & \text{IH} \\ = \mathcal{W}_n(e \rhd M_1) \uplus \mathcal{W}_n(N) & \end{array}$$

∎

As an immediate corollary of Lem. C.4.3 we obtain a proof of Lem. 5.7.7 ($\mathcal{W}_n((\lambda a : A.M)\, N) \succeq \mathcal{W}_n(M^a_{N,t})$).

Lemma C.4.4 $\mathcal{W}^t(N^u_{\Sigma'.(M,t,e)}) \leq \mathcal{W}^t(N)$.

Lemma C.4.5 If M has no occurrences of the "!" term constructor, then $\mathcal{W}_n(N^u_{\Sigma.(M,t,e)}) \preceq \mathcal{W}_n(N)$.

Proof. By induction on N using Lem. C.4.1 and Lem. C.4.4.

- $N = a$. $\mathcal{W}_n(N^u_{\Sigma.(M,t,e)}) = \langle\!\langle\ \rangle\!\rangle = \mathcal{W}_n(N)$.
- $N = \lambda a : A.N_1$.

$$\begin{array}{ll} \mathcal{W}_n((\lambda a : A.N_1)^u_{\Sigma.(M,t,e)}) & \\ = \mathcal{W}_n(\lambda a : A.{N_1}^u_{\Sigma.(M,t,e)}) & \\ = \mathcal{W}_n({N_1}^u_{\Sigma.(M,t,e)}) & \\ \preceq \mathcal{W}_n(N_1) & \text{IH} \\ = \mathcal{W}_n(\lambda a : A.N_1) & \end{array}$$

- $N = N_1\,N_2$.

$$\begin{array}{ll}
\mathcal{W}_n((N_1\,N_2)^u_{\Sigma.(M,t,e)}) & \\
= \mathcal{W}_n(N_1{}^u_{\Sigma.(M,t,e)}\,N_2{}^u_{\Sigma.(M,t,e)}) & \\
= \mathcal{W}_n(N_1{}^u_{\Sigma.(M,t,e)}) \uplus \mathcal{W}_n(N_2{}^u_{\Sigma.(M,t,e)}) & \\
\preceq \mathcal{W}_n(N_1) \uplus \mathcal{W}_n(N_2) & \text{IH} \\
= \mathcal{W}_n(N_1\,N_2) &
\end{array}$$

- $N = \langle v;\sigma\rangle$.

$$\begin{array}{l}
\mathcal{W}_n(\langle v;\sigma\rangle^u_{\Sigma.(M,t,e)}) \\
= \mathcal{W}_n(\langle v;\sigma\rangle)
\end{array}$$

- $N = \langle u;\sigma\rangle$.

$$\begin{array}{ll}
\mathcal{W}_n(\langle u;\sigma\rangle^u_{\Sigma.(M,t,e)}) & \\
= \mathcal{W}_n(e\sigma \rhd M\sigma) & \\
= \mathcal{W}_n(M\sigma) & \\
= \langle\!\langle\ \rangle\!\rangle & \text{L. C.4.1} \\
= \mathcal{W}_n(\langle u;\sigma\rangle) &
\end{array}$$

- $N = !^{\Sigma_1}_{e_1} N_1$. Let $m \stackrel{\text{def}}{=} n * \mathcal{W}^t(N_1{}^u_{\Sigma.(M,t,e)})$.

$$\begin{array}{ll}
\mathcal{W}_n((!^{\Sigma_1}_{e_1} N_1)^u_{\Sigma.(M,t,e)}) & \\
= \mathcal{W}_n(!^{\Sigma_1}_{e_1{}^u_{\Sigma.t}} N_1{}^u_{\Sigma.(M,t,e)}) & \\
= m \uplus \mathcal{W}_m(N_1{}^u_{\Sigma.(M,t,e)}) & \\
\preceq n * \mathcal{W}^t(N_1) \uplus \mathcal{W}_m(N_1{}^u_{\Sigma.(M,t,e)}) & \text{L. C.4.4} \\
\preceq n * \mathcal{W}^t(N_1) \uplus \mathcal{W}_m(N_1) & \text{IH} \\
\preceq n * \mathcal{W}^t(N_1) \uplus \mathcal{W}_{n*\mathcal{W}^t(N_1)}(N_1) & \text{L. C.4.2} \\
= \mathcal{W}_n(!^{\Sigma_1}_{e_1} N_1) &
\end{array}$$

- $N = \mathsf{let}\, u : A[\Sigma] = N_1 \,\mathsf{in}\, N_2$. Similar to the case of application.
- $N = \alpha\theta$.

$$\begin{array}{ll}
\mathcal{W}_n((\alpha\theta)^u_{\Sigma.(M,t,e)}) & \\
= \mathcal{W}_n(\alpha(\theta^u_{\Sigma.(M,t,e)})) & \\
= \biguplus_{i\in 1..10} \mathcal{W}_n(\theta(c_i)^u_{\Sigma.(M,t,e)}) & \\
\preceq \biguplus_{i\in 1..10} \mathcal{W}_n(\theta(c_i)) & \text{IH*10} \\
= \mathcal{W}_n(\alpha\theta) &
\end{array}$$

- $N = e_1 \rhd N_1$.

$$\begin{array}{ll}
\mathcal{W}_n((e_1 \rhd N_1)^u_{\Sigma.(M,t,e)}) & \\
= \mathcal{W}_n(e_1{}^u_{\Sigma.t} \rhd N_1{}^u_{\Sigma.(M,t,e)}) & \\
= \mathcal{W}_n(N_1{}^u_{\Sigma.(M,t,e)}) & \\
\preceq \mathcal{W}_n(N_1) & \text{IH} \\
= \mathcal{W}_n(e_1 \rhd N_1) &
\end{array}$$

∎

Proof of Lem. 5.7.8:
Suppose M has no occurrences of the modal type constructor. Then $\mathcal{W}_n(\mathsf{let}\, u : A[\Sigma] = !^{\Sigma}_{e} M \,\mathsf{in}\, N) \succ \mathcal{W}_n(\mathfrak{bb}(u^{A[\Sigma]}.s, \Sigma.t) \rhd N^u_{\Sigma.(M,t,e)})$.

Proof. On the one hand:

$$\begin{array}{l} \mathcal{W}_n(\mathsf{let}\, u : A[\Sigma] = !^{\Sigma}_{e} M \,\mathsf{in}\, N) \\ = \mathcal{W}_n(!^{\Sigma}_{e} M) \uplus \mathcal{W}_n(N) \\ = n * \mathcal{W}^t(M) \uplus \mathcal{W}_{n*\mathcal{W}^t(M)}(M) \uplus \mathcal{W}_n(N) \end{array}$$

On the other hand:

$$\begin{array}{l} \mathcal{W}_n(\mathfrak{bb}(u^{A[\Sigma]}.s, \Sigma.t) \rhd N^u_{\Sigma.(M,t,e)}) \\ = \mathcal{W}_n(N^u_{\Sigma.(M,t,e)}) \end{array}$$

By Lem. C.4.5, we conclude.

∎

Lemma C.4.6 $m < n$ implies $\mathcal{W}_m(\mathcal{C}[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]) \preceq \mathcal{W}_n(\mathcal{C}[\alpha\theta])$.

Proof. By induction on $\mathcal{C}$.

- $\mathcal{C} = \Box$. We reason as follows:

$$\begin{array}{l} \mathcal{W}_m(\mathsf{ti}(\theta^w, \alpha) \rhd e\theta) \\ = \mathcal{W}_m(e\theta) \\ \preceq \mathcal{W}_n(\alpha\theta) \end{array}$$

The justification of the last step in our reasoning is as follows. Note that $\mathcal{W}_m(e\theta) = \biguplus_{i\in 1..10} \mathcal{W}_m(\theta(c_i))^{p_i}$, where p_i is the number of occurrences of the constructor c_i in e. Also, $\mathcal{W}_n(\alpha\theta) = \biguplus_{i\in 1..10} \mathcal{W}_n(\theta(c_i))$. Thus for each $i \in 1..10$ we can "pair" $\mathcal{W}_n(\theta(c_i))$ with $\mathcal{W}_m(\theta(c_i))^{p_i}$. For each c_i we have two cases:
  - If $\theta(c_i)$ has no occurrences of the modal term constructor "!", then $\mathcal{W}_m(e\theta) \preceq \mathcal{W}_n(\alpha\theta)$ by Lem. C.4.1.
  - If $\theta(c_i)$ has at least one occurrence of the modal term constructor "!", then $\mathcal{W}_m(e\theta) \preceq \mathcal{W}_n(\alpha\theta)$ by Lem. C.4.2(2) and the definition of the multiset extension of an ordering.
- $\mathcal{C} = \lambda a : A.\mathcal{C}'$. Similar to the case for application developed below.
- $\mathcal{C} = \mathcal{C}'\ M$. We reason as follows:

$$\begin{array}{ll} \mathcal{W}_m(\mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]\ M) & \\ = \mathcal{W}_m(\mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]) \uplus \mathcal{W}_m(M) & \\ \preceq \mathcal{W}_m(\mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]) \uplus \mathcal{W}_n(M) & L.\ C.4.2 \\ \preceq \mathcal{W}_n(\mathcal{C}'[\alpha\theta]) \uplus \mathcal{W}_n(M) & \text{IH} \\ = \mathcal{W}_n(\mathcal{C}'[\alpha\theta]\ M) & \end{array}$$

- $\mathcal{C} = M\ \mathcal{C}'$. Similar to the previous case.
- $\mathcal{C} = \mathsf{let}\, u : A[\Sigma] = \mathcal{C}' \,\mathsf{in}\, M_1$. We reason as follows:

$$\begin{array}{ll} \mathcal{W}_m(\mathsf{let}\, u : A[\Sigma] = \mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta] \,\mathsf{in}\, M_1) & \\ = \mathcal{W}_m(\mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]) \uplus \mathcal{W}_m(M_1) & \\ \preceq \mathcal{W}_m(\mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]) \uplus \mathcal{W}_n(M_1) & L.\ C.4.2 \\ \preceq \mathcal{W}_n(\mathcal{C}'[\alpha\theta]) \uplus \mathcal{W}_n(M_1) & \text{IH} \\ = \mathcal{W}_n(\mathsf{let}\, u : A[\Sigma] = \mathcal{C}'[\alpha\theta] \,\mathsf{in}\, M_1) & \end{array}$$

- $\mathcal{C} = \mathsf{let}\, u : A[\Sigma] = M_1 \,\mathsf{in}\, \mathcal{C}'$. Similar to the previous case.
- $\mathcal{C} = e_1 \rhd \mathcal{C}'$.

$$\begin{array}{ll} \mathcal{W}_m(e_1 \rhd \mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]) & \\ = \mathcal{W}_m(\mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]) & \\ \preceq \mathcal{W}_n(\mathcal{C}'[\alpha\theta]) & \text{IH} \\ = \mathcal{W}_n(e_1 \rhd \mathcal{C}'[\alpha\theta]) & \end{array}$$

∎

Proof of Lem. 5.7.9 ($\mathcal{W}_n(!^{\Sigma}_{e}\mathcal{C}[\alpha\theta]) \succ \mathcal{W}_n(!^{\Sigma}_{e}\mathcal{C}[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta])$).

Proof. By induction on $\mathcal{C}$.

- $\mathcal{C} = \Box$. Let $m \overset{\text{def}}{=} n * \mathcal{W}^t(\alpha\theta)$ and $m' \overset{\text{def}}{=} n * \mathcal{W}^t(\mathsf{ti}(\theta^w, \alpha) \rhd e\theta)$. Note that $m' < m$. We reason as follows:

$$\begin{array}{ll}
& \mathcal{W}_n(!^{\Sigma}_e \alpha\theta) \\
= & m \uplus \mathcal{W}_m(\alpha\theta) \\
= & n * 2 \uplus \mathcal{W}_m(\alpha\theta) \\
\succ & n \uplus \mathcal{W}_m(\alpha\theta) \\
= & n * 1 \uplus \mathcal{W}_m(\alpha\theta) \\
= & n * \mathcal{W}^t(\mathsf{ti}(\theta^w, \alpha) \rhd e\theta) \uplus \mathcal{W}_m(\alpha\theta) \\
\succeq & n * \mathcal{W}^t(\mathsf{ti}(\theta^w, \alpha) \rhd e\theta) \uplus \mathcal{W}_{m'}(\mathsf{ti}(\theta^w, \alpha) \rhd e\theta) \; L.\ C.4.6 \\
= & \mathcal{W}_n(!^{\Sigma}_e \mathsf{ti}(\theta^w, \alpha) \rhd e\theta)
\end{array}$$

- $\mathcal{C} = \lambda a : A.\mathcal{C}'$. Similar to the case for application developed below.
- $\mathcal{C} = \mathcal{C}'\ M$. Let $m \overset{\text{def}}{=} n * \mathcal{W}^t(\mathcal{C}'[\alpha\theta]\, M)$ and $m' \overset{\text{def}}{=} n * \mathcal{W}^t(\mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]\, M)$. Note that $m' < m$. We reason as follows:

$$\begin{array}{lll}
& \mathcal{W}_n(!^{\Sigma}_e(\mathcal{C}'[\alpha\theta]\, M)) & \\
= & m \uplus \mathcal{W}_m(\mathcal{C}'[\alpha\theta]\, M) & \\
= & m \uplus \mathcal{W}_m(\mathcal{C}'[\alpha\theta]) \uplus \mathcal{W}_m(M) & \\
\succ & m' \uplus \mathcal{W}_m(\mathcal{C}'[\alpha\theta]) \uplus \mathcal{W}_m(M) & \\
\succeq & m' \uplus \mathcal{W}_{m'}(\mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]) \uplus \mathcal{W}_m(M) & L.\ C.4.6 \\
\succeq & m' \uplus \mathcal{W}_{m'}(\mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]) \uplus \mathcal{W}_{m'}(M) & L.\ C.4.2 \\
= & m' \uplus \mathcal{W}_{m'}(\mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]\, M) & \\
= & \mathcal{W}_n(!^{\Sigma}_e \mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]\, M) &
\end{array}$$

- $\mathcal{C} = M\,\mathcal{C}'$. Similar to the previous case.
- $\mathcal{C} = \mathsf{let}\, u : A[\Sigma] = \mathcal{C}'\ \mathsf{in}\ M_1$. Similar to the case for application.
- $\mathcal{C} = \mathsf{let}\, u : A[\Sigma] = M_1\ \mathsf{in}\ \mathcal{C}'$. Similar to the case for application.
- $\mathcal{C} = e_1 \rhd \mathcal{C}'$. Let $m \overset{\text{def}}{=} n * \mathcal{W}^t(e_1 \rhd \mathcal{C}'[\alpha\theta])$ and $m' \overset{\text{def}}{=} n * \mathcal{W}^t(e_1 \rhd \mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta])$. Note that $m' < m$. We reason as follows:

$$\begin{array}{lll}
& \mathcal{W}_n(!^{\Sigma}_e(e_1 \rhd \mathcal{C}'[\alpha\theta])) & \\
= & m \uplus \mathcal{W}_m(e_1 \rhd \mathcal{C}'[\alpha\theta]) & \\
= & m \uplus \mathcal{W}_m(\mathcal{C}'[\alpha\theta]) & \\
\succ & m' \uplus \mathcal{W}_m(\mathcal{C}'[\alpha\theta]) & \\
\succeq & m' \uplus \mathcal{W}_{m'}(\mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]) & L.\ C.4.6 \\
= & m' \uplus \mathcal{W}_{m'}(e_1 \rhd \mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta]) & \\
= & \mathcal{W}_n(!^{\Sigma}_e(e_1 \rhd \mathcal{C}'[\mathsf{ti}(\theta^w, \alpha) \rhd e\theta])) &
\end{array}$$

∎

C.5 Termination of Permutation Schemes

The encoding of the permutation reduction rules adapted for the AProVe tool. The output, with the result that it indeed could be proven SN may be consulted at http://tpi.blog.unq.edu.ar/~ebonelli/SNofPermRed.pdf.

```
(VAR m n e f e1 e2 e3 e4 e5)
(RULES
        app(trail(e,m),n)      -> trail(appCong(e,rfl),app(m,n))
        app(m,trail(e,n))      -> trail(appCong(rfl,e),app(m,n))
        abs(trail(e,m))       -> trail(absCong(e),abs(m))
        let(trail(e,m),n)  -> trail(letCong(e,rfl),let(m,n))
        let(m,trail(e,n))  -> trail(letCong(rfl,e),let(m,n))
        auditedUnit(e,trail(f,m))     -> auditedUnit(trn(e,f),m)
        trail(e,trail(f,m))    -> trail(trn(e,f),m)
        trn(appCong(e1,e2),appCong(e3,e4))   -> appCong(trn(e1,e3),trn(e2,e4))
        trn(absCong(e1),absCong(e2))         -> absCong(trn(e1,e2))
        trn(letCong(e1,e2),letCong(e3,e4)) -> letCong(trn(e1,e3),trn(e2,e4))
trn(rfl,e) -> e
        trn(e,rfl) -> e
trn(trn(e1,e2),e3) -> trn(e1,trn(e2,e3))
trn(appCong(e1,e2),trn(appCong(e3,e4),e5))   -> trn(appCong(trn(e1,e3),trn(e2,e4)),e5)
trn(absCong(e1),trn(absCong(e2),e3))         -> trn(absCong(trn(e1,e2)),e3)
trn(letCong(e1,e2),trn(letCong(e3,e4),e5)) -> trn(letCong(trn(e1,e3),trn(e2,e4)),e5)
)
```

References

1. Nist special publications. generally accepted principles and practices for securing information technology systems., September 1996. 1
2. Iso/iec. common criteria for information technology security evaluation., 2004. 1
3. Programming language popularity. http://www.langpop.com/, Retrieved 2009-01-16. 2009. 2.2
4. M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL*, pages 147–160, 1999. 2.1
5. M. Abadi and C. Fournet. Access control based on execution history. In *NDSS*. The Internet Society, 2003. 5.3
6. Ada. Ada-europe. http://www.ada-europe.org/. 2.2
7. J. Agat. Transforming out timing leaks. In *POPL*, pages 40–53, 2000. 2.1
8. J. Alt and S. Artemov. Reflective λ-calculus. In *Proceedings of the Dagstuhl-Seminar on Proof Theory in CS*, volume 2183 of *LNCS*, 2001. 5.8.1
9. S. Artemov. Operational modal logic. Technical Report MSI 95-29, Cornell University, 1995. 1.1, 5, 5.1, 5.8.1
10. S. Artemov. Explicit provability and constructive semantics. *Bulletin of Symbolic Logic*, 7(1):1–36, 2001. 1.1, 5, 5.1, 5.8.1
11. S. Artemov. Justification logic. In S. Hölldobler, C. Lutz, and H. Wansing, editors, *JELIA*, volume 5293 of *Lecture Notes in Computer Science*, pages 1–4. Springer, 2008. 1.1, 5, 5.1, 5.8.1
12. S. Artemov and E. Bonelli. The intensional lambda calculus. In *LFCS*, volume 4514 of *LNCS*, pages 12–25. Springer, 2007. 5.2, 5.2, 5.8.1
13. A. Askarov and A. C. Myers. Attacker control and impact for confidentiality and integrity. *Logical Methods in Computer Science*, 7(3), 2011. 4.5.2
14. L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker. A domain and type enforcement unix prototype. *Computing Systems*, 9(1):47–83, 1996. 1
15. T. Ball. What's in a region?: or computing control dependence regions in near-linear time for reducible control flow. *ACM Lett. Program. Lang. Syst.*, 2(1-4):1–16, 1993. 3.3.5
16. A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of the Fifteenth IEEE Computer Security Foundations Workshop (CSFW)*, pages 253–267. IEEE Computer Society Press, 2002. 3.1
17. A. Banerjee and D. A. Naumann. Using access control for secure information flow in a java-like language. In *CSFW*, pages 155–169. IEEE Computer Society, 2003. 2.1, 3.7.2
18. A. Banerjee and D. A. Naumann. History-based access control and secure information flow. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 27–48. Springer, 2004. 5.3.1, 3
19. A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005. Special Issue on Language-Based Security. 1, 2.1, 3.1, 3.6
20. H. Barendregt. *Handbook of Logic in Computer Science*, chapter Lambda Calculi with Types. Oxford University Press, 1992. 5.8.1
21. G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. *Journal of Computer Languages, Systems and Structures*, 2005. 3.3.2, 3.7.2

22. G. Barthe, S. Cavadini, and T. Rezk. Tractable enforcement of declassification policies. *Computer Security Foundations Symposium, IEEE*, 0:83–97, 2008. 4.5.2
23. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011. 3.7.2
24. G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. In *Proc. of 16th European Symposium on Programming (ESOP'07)*, Lecture Notes in Computer Science. Springer-Verlag, 2007. to appear. 3.1, 3.1, 3.1, 3.3.2, 3.3.5
25. G. Barthe and T. Rezk. Non-interference for a JVM-like language. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 103–112, New York, NY, USA, 2005. ACM Press. 2.1, 3.1, 3.1, 3.7.2
26. G. Barthe, T. Rezk, and D. A. Naumann. Deriving an information flow checker and certifying compiler for java. In *S&P*, pages 230–242. IEEE Computer Society, 2006. 3.1, 3.1, 3.3.2, 3.7.2
27. G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *Proc. of the 12th ESORICS*, LNCS. Springer-Verlag, 2007. To appear. 3.7.1, 3.7.2, 6.2
28. F. Bavera and E. Bonelli. Type-based information flow analysis for bytecode languages with variable object field policies. In *SAC'08, Proceedings of the 23rd Annual ACM Symposium on Applied Computing. Software Verification Track*, 2008. 1.1, 2.1
29. F. Bavera and E. Bonelli. Robust declassification for bytecode. In *V Congreso Iberoamericano de Seguridad Informática (CIBSI'09)*, 2009. 1.1
30. F. Bavera and E. Bonelli. Justification logic and history based computation. In *7th International Colloquium on Theoretical Aspects of Computing (ICTAC 2010)*, 2010. 1.1
31. C. Bernardeschi and N. D. Francesco. Combining abstract interpretation and model checking for analysing security properties of java bytecode. In A. Cortesi, editor, *VMCAI*, volume 2294 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2002. 3.7.2
32. C. Bernardeschi, N. D. Francesco, G. Lettieri, and L. Martini. Checking secure information flow in java bytecode by code transformation and standard bytecode verification. *Softw., Pract. Exper.*, 34(13):1225–1255, 2004. 3.7.2
33. C. Bernardeschi, N. D. Francesco, and L. Martini. Efficient bytecode verification using immediate postdominators in control flow graphs (extended abstract). 3.3.5
34. K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, apr 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.). 2.1
35. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J. Lanet. Checking secure interactions of smart card applets. *Journal of Computer Security*, 1(10):369–398, 2002. 3.7.2
36. M. Bishop. *Introduction to Computer Security*. Addison-Wesley Professional, 2004. 1
37. P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press, 2001. 5.1
38. E. Bonelli, A. Compagnoni, and R. Medel. Information-flow analysis for typed assembly languages with polymorphic stacks. In *Proc. of Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS'05)*, volume 3956 of *LNCS*. Springer-Verlag, 2005. 2.1, 3.1, 3.1, 3.7.2
39. E. Bonelli and F. Feller. The logic of proofs as a foundation for certifying mobile computation. In *LFCS*, volume 5407 of *LNCS*, pages 76–91. Springer, 2009. 5.1, 5.8.1
40. E. Bonelli and E. Molinari. Compilación de programas seguros. In *Proc. of Simposio Argentino de Ingeniería de Software (ASSE'09), 38 JAIIO*, 2009. 3.7.2
41. G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theor. Comput. Sci.*, 281(1-2):109–130, 2002. 2.1
42. V. Brezhnev. On the logic of proofs. In *Proceedings of the Sixth ESSLLI Student Session*, pages 35–45, 2001. 5.8.1
43. W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In P. Ning, P. F. Syverson, and S. Jha, editors, *ACM Conference on Computer and Communications Security*, pages 39–50. ACM, 2008. 1
44. J. Chrzaszcz, P. Czarnik, and A. Schubert. A dozen instructions make java bytecode. In D. Pichardie, editor, *Bytecode'2010*, ENTCS, 2010. 2.2
45. E. S. Cohen. Information transmission in computational systems. In *SOSP*, pages 133–139, 1977. 2.1.1, 3.7.2
46. T. C. Community. The coq proof assistant: Reference manual. http://coq.inria.fr/refman/. 6.2

47. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977. 3.7.2
48. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Proc. 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer-Verlag, 2005. 3.7.2
49. P. David. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*. PhD thesis, Université Rennes 1, 2005. In french. 6.2
50. R. Davies and F. Pfenning. A modal analysis of staged computation. In *23rd POPL*, pages 258–270. ACM Press, 1996. 5.1, 5.2, 5.8.1
51. R. Davies and F. Pfenning. A judgmental reconstruction of modal logic. *Journal of MSCS*, 11:511–540, 2001. 5.2, 1, 5.8.1
52. R. Davies and F. Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, May 2001. 5.1, 5.2, 1, 5.8.1
53. D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976. 1, 3.7.2
54. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977. 1, 3.7.2
55. C. Fournet and A. D. Gordon. Stack inspection: theory and variants. In *POPL*, pages 307–318, 2002. 1
56. S. Freund and J. Mitchell. A type system for the java bytecode language and verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, 2003. 3.7.2
57. S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In R. Cousot, editor, *Proc. of the Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362, Paris, France, January 2005. Springer-Verlag. 3.7.2
58. D. Ghindici. *Information flow analysis for embedded systems: from practical to theoretical aspects*. PhD thesis, Universite des Sciences et Technologies de Lille, 2008. 2.1, 3.7.2
59. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with aprove. In *RTA*, volume 3091 of *LNCS*, pages 210–220. Springer, 2004. 5.6.1
60. J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989. 5.1
61. V. D. Gligor. A Guide To Understanding Covert Channel Analysis of Trusted Systems. Technical report, NSA/NCSC Rainbow, National Computer Security Center, 1993. 2.1
62. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April, 1982. 2.1.1, 3.6, 3.7.2
63. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java language specification*. Addison Wesley, third edition. edition, 20005. 2.2
64. N. Heintze and J. G. Riecke. The slam calculus: Programming with secrecy and integrity. In *POPL*, pages 365–377, 1998. 2.1
65. S. Hunt and D. Sands. On flow-sensitive security types. In *POPL'06, Proceedings of the 33rd Annual. ACM SIGPLAN - SIGACT. Symposium. on Principles of Programming Languages*, January 2006. 2.1.2, 3.7.2
66. L. Jia and D. Walker. Modal proofs as distributed programs (extended abstract). In D. A. Schmidt, editor, *ESOP*, volume 2986 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 2004. 5.8.1
67. L. Jiang, L. Ping, and X. Pan. Combining robust declassification and intransitive noninterference. In *IMSCCS*, pages 465–471. IEEE, 2007. 4.5.1, 4.5.2, 6.2
68. N. Kobayashi and K. Shirane. Type-based information flow analysis for a low-level language. *Proceedings of the 3rd Asian Workshop on Programming Languages and Systems*, 2002. Computer Software 20(2), Iwanami Press, pp.2-21, 2003 (in Japanese). The English version appeared in informal Proceedings of the 3rd Asian Workshop on Programming Languages and Systems (APLAS'02). 2.1, 3.3.5, 3.7.2
69. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999. 2.1
70. D. Kozen. Language-based security. In M. Kutylowski, L. Pacholski, and T. Wierzbicki, editors, *MFCS*, volume 1672 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 1999. 2.1
71. B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973. 2.1, 6.2
72. G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007. 1
73. X. Leroy. Bytecode verification for java smart card. *Software Practice and Experience*, 32:319–340, 2002. 3.1

74. P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In J. Palsberg and M. Abadi, editors, *POPL*, pages 158–170. ACM, 2005. 2.1
75. T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification*. Addison Wesley, 1999. 1, 2.2, 3.1, 3.4.1, 3.5.1, 4.1
76. P. Martin-Löf. On the meaning of the logical constants and the justifications of the logical laws. *Nordic J. of Philosophical Logic 1*, 1:11–60, 1996. 5.2, 5.8.1
77. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. 3.7.2
78. K. Miyamoto and A. Igarashi. A modal foundation for secure information flow. In *Workshop on Foundations of Computer Security (FCS'04)*, pages 187–203, 2004. 5.1
79. J. Moody. Logical mobility and locality types. In S. Etalle, editor, *LOPSTR*, volume 3573 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2004. 5.1, 5.8.1
80. G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1999. 2.1, 3.1
81. A. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. http://www.cs.cornell.edu/jif/, 2001. Software release. 6.2
82. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages (POPL'99)*, pages 228–241, 1999. ACM Press. 1, 2.1, 3.7.2
83. A. C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Laboratory of Computer Science, MIT, 1999. 2.1
84. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *In Proceedings of 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, 1997. 1, 3.7.2
85. A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *CSFW*, pages 172–186. IEEE Computer Society, 2004. 2.1, 4, 4.1, 4.2, 4.5.1, 4.5.2, 6.2
86. S. K. Nair. *Remote Policy Enforcement Using Java Virtual Machine*. PhD thesis, Vrije Universiteit, 2010. 2.1
87. S. K. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electr. Notes Theor. Comput. Sci.*, 197(1):3–16, 2008. 2.1
88. A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3), 2008. 5.8.1
89. G. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1998. 2.1
90. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. 3.3.5, 3.3.5, 3.3.5
91. F. Pfenning and H.-C. Wong. On a modal λ-calculus for S4. In *Proceedings of the Eleventh Conference on Mathematical Foundations of Programming Semantics (MFPS'95)*, 1995. Available as Electronic Notes in Theoretical Computer Science, Volume 1, Elsevier. 1
92. V. Ranganath, T. Amtoft, A. Banerjee, M. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. Tech report, SAnToS Lab., Kansas State University, 2004. 3.3.5
93. J. Rehof and T. Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35(2–3):191–221, 1999. 3.3.5
94. T. Rezk. *Verification of confidentiality policies for mobile code*. PhD thesis, Université de Nice and INRIA Sophia Antipolis, 2006. 1, 2.1, 3.1, 3.22, 3.5.2, 3.7.2, 3.7.2
95. A. Russo. *Language Support for Controlling Timing-Based Covert Channels*. PhD thesis, Chalmers University of Technology, 2008. 6.2
96. A. Russo, J. Hughes, D. A. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In M. Okada and I. Satoh, editors, *ASIAN*, volume 4435 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2006. 2.1
97. A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler in the presence of synchronization. *J. Log. Algebr. Program.*, 78(7):593–618, 2009. 2.1
98. A. Sabelfeld and D. Hedin. A perspective on information-flow control. *Proceedings of the 2011 Marktoberdorf Summer School, IOS Press*, 2011. 3.7.2
99. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003. 1
100. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW*, pages 200–214, 2000. 2.1

101. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW-18 2005*, pages 255–269. IEEE Computer Society, 2005. 2.1.3
102. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 2007. To appear. 2.1.3, 4.5.2
103. Scala. The scala programming language. http://www.scala-lang.org/. 2.2
104. F. B. Schneider, J. G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 86–101, London, UK, 2001. Springer-Verlag. 2.1
105. V. Simonet and I. Rocquencourt. Flow caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*, pages 152–165, 2003. 1, 2.1
106. G. Smith. Principles of secure information flow analysis. In *Malware Detection*, pages 297–307. Springer-Verlag, 2007. 2.1
107. G. Smith and D. M. Volpano. Confinement properties for multi-threaded programs. *Electr. Notes Theor. Comput. Sci.*, 20, 1999. 2.1
108. Q. Sun. *Constraint-Based Modular Secure Information Flow Inference for Object-Oriented Programs*. PhD thesis, Stevens Institute of Technology, 2008. 1
109. Q. Sun, D. Naumann, and A. Banerjee. Constraint-based secure information flow inference for a java-like language. Tech Report 2004-2, Kansas State University CIS, 2005. 2.1
110. I. "Sun Microsystems. http://java.sun.com/products/javacard/, 2008. 3.1
111. J. Svenningsson and D. Sands. Specification and verification of side channel declassification. In P. Degano and J. D. Guttman, editors, *Formal Aspects in Security and Trust*, volume 5983 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2009. 3.7.2, 6.2
112. W. Taha and T. Sheard. Multi-stage programming. In *ICFP*, page 321, 1997. 5.1
113. T. M. VII, K. Crary, and R. Harper. Distributed control flow with classical modal logic. In C.-H. L. Ong, editor, *CSL*, volume 3634 of *Lecture Notes in Computer Science*, pages 51–69. Springer, 2005. 5.1, 5.8.1
114. T. M. VII, K. Crary, and R. Harper. Type-safe distributed programming with ml5. In G. Barthe and C. Fournet, editors, *TGC*, volume 4912 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2007. 5.8.1
115. T. M. VII, K. Crary, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. In *LICS*, pages 286–295. IEEE Computer Society, 2004. 5.1, 5.8.1
116. D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of TAPSOFT'97*, volume 1214 of *LNCS*, pages 607–621, 1997. 1, 2.1, 3.7.2
117. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996. 1, 2.1, 2.1.1, 3.7.2
118. D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996. 3.1, 5.3.1
119. D. M. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *CSFW*, pages 156–169. IEEE Computer Society, 1997. 2.1
120. D. M. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(1), 1999. 2.1
121. D. Walker. A type system for expressive security policies. In *POPL*, pages 254–267, 2000. 1
122. J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999. 3.7.2
123. P. Wickline, P. Lee, F. Pfenning, and R. Davies. Modal types as staging specifications for run-time code generation. *ACM Comput. Surv.*, 30(3es):8, 1998. 5.1
124. D. Yu and N. Islam. A typed assembly language for confidentiality. In *2006 European Symposium on Programming (ESOP'06)*. LNCS Vol. 3924., 2006. 2.1, 3.7.2
125. S. Zdancewic. A type system for robust declassification. *Proceedings of the Nineteenth Conference on the Mathematical Foundations of Programming Semantics. Electronic Notes in Theoretical Computer Science*, 2003. 4.5.2
126. S. Zdancewic. Challenges for information-flow security. In *In Proc. Programming Language Interference and Dependence (PLID'04)*, 2004. 2.1.3
127. S. Zdancewic and A. C. Myers. Robust declassification. In *CSFW-14 2001*. IEEE Computer Society, 2001. 4.5, 4.5.2, 6.1