Università Ca'Foscari Venezia

Scuola Dottorale di Ateneo
Graduate School

Dottorato di ricerca
in Informatica
Ciclo XXV
Anno di discussione 2013

Static Verification and Enforcement of Authorization Policies

SETTORE SCIENTIFICO DISCIPLINARE DI AFFERENZA: INF/01
Tesi di Dottorato di Stefano Calzavara, matricola 801411

Coordinatore del Dottorato	Tutore del Dottorando
Prof. Riccardo Focardi	Prof. Michele Bugliesi

Università Ca' Foscari di Venezia

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

Static Verification and Enforcement of Authorization Policies

Stefano Calzavara

Supervisor
Prof. Michele Bugliesi

Referee
Prof. Pierpaolo Degano

Referee
Dr. Cédric Fournet

January 28, 2013

Author's Web Page: www.dais.unive.it/~calzavara

Author's e-mail: calzavara@dais.unive.it

Author's address:

Dipartimento di Informatica
Università Ca' Foscari di Venezia
Via Torino, 155
30172 Venezia Mestre – Italia
tel. +39 041 2348411
fax. +39 041 2348419
web: `http://www.dsi.unive.it`

To my father Angelo (Tito)

Abstract

This thesis addresses the problem of statically verifying and enforcing authorization policies in a range of different settings.

The problem of policy *verification* consists in trying to fill in the gap between a high-level policy specification and its low-level expected security requirements: this problem can be effectively addressed through static model-checking techniques, which allow to assess the robustness of a given authorization policy way before its actual enforcement. The thesis introduces the formal machinery to perform such analysis on role-based access control policies for `grsecurity`, an extension for the Linux kernel aimed at enhancing the operating system security. An implementation of the analysis is discussed, along with a preliminary investigation about the security of existing `grsecurity` access control policies.

Policy *enforcement* is an orthogonal issue to policy verification: namely, given a policy specification, the challenge is understanding if a specific system implementation does comply with the policy. A recently emerging trend consists in developing static techniques for proving policy compliance of application code. The contributions of the thesis in such research line are twofold:

- First, a refinement type system aimed at enforcing *resource-aware* authorization policies, expressed in a fragment of intuitionistic affine logic, is developed for RCF, a concurrent lambda-calculus with message-passing primitives. Extending existing proposals to such sub-structural setting proves challenging, but rewarding: affine logic, in fact, accounts for very natural specifications of complex resourceful policies, which can be enforced with only limited effort by the programmer.

- Second, a static type-based analysis technique is devised for a simple abstraction of the widespread Android platform for mobile devices. The goal is enforcing a more robust access control policy, by programmatically preventing a subtle vulnerability related to the interplay of the communication paradigm with the underlying permission system, which allows malicious components to escalate privileges. The technique is proved sound. The challenges related to the implementation of the framework, or any other type-based analysis technique for Android applications, are discussed in detail.

Acknowledgments

Many thanks are due to my supervisor Michele Bugliesi. Tough incredibly busy, Michele always tried his best to be a caring and friendly guide during all these three years. His deep understanding of the problems proved invaluably helpful in improving my scientific background.

My gratitude goes to the reviewers of this thesis, prof. Pierpaolo Degano and dr. Cédric Fournet. I'm very proud of them accepting this role and I thank them for the time spent in reading this work.

I owe much to many people in Ca' Foscari. I would like to thank Damiano Macedonio, who co-supervised my M.Sc. thesis. Damiano is an enthusiastic teacher and a friend: many discussions with him surely played a prominent role in convincing me to pursue my Ph.D. I am deeply indebted to Riccardo Focardi, who gave me the chance to work with him on the interesting and successful research on `grsecurity`. I would like also to thank Paolo Modesti, Alvise Spanò and Marco Squarcina for their precious collaboration, their patience in explaining technological problems, and many funny moments shared together.

Moving outside from Venice, I'm really grateful to Matteo Maffei: without his assistance and his optimistic views, all the work on RCF would have never been possible. I want also to thank Fabienne Eigner for her invaluable help on a number of technical problems, which arose throughout the proofs of our type system.

Many thanks to my family: my mother Ilva, my brother Riccardo and my father Angelo, who left just before I started my Ph.D and whom I dedicate this thesis. Their unconditional support during my studies was a fundamental building block for my scientific formation.

I want also to mention some friends, in particular: Dario, for our long-standing friendship; Giacomo, for his enthusiastic approach to life; Daniele, Fabio and Raffaele, for too many moments shared together. Finally, a special thought goes to Valentina: without her, these three years would have been much harder.

Contents

List of Tables

Preface

The work presented in this thesis is based on some published research papers written during my Ph.D. studies in Computer Science at Università Ca' Foscari Venezia from January 2010 to December 2012.

Chapter 1 is a joint work with Michele Bugliesi, Riccardo Focardi and Marco Squarcina presented at the *25th IEEE Computer Security Foundations Symposium* in June 2012 [22].

Chapter 2 is the outcome of a long research program carried out in collaboration with Michele Bugliesi, Fabienne Eigner and Matteo Maffei. Preliminary results were presented at the *24th IEEE Computer Security Foundations Symposium* in June 2011 [19] and at the *7th International Symposium on Trustworthy Global Computing* in September 2012 [20]. The contents of the chapter significantly extend such previous work and have been accepted for publication at the *2nd Conference on Principles of Security and Trust*, to be held in March 2013 [21].

Finally, Chapter 4 is an on-going joint work with Michele Bugliesi and Alvise Spanò. At the time of writing we are still performing our experiments and completing some formal details. Hopefully, we should be able to substantiate our claims with experimental results and submit a paper soon.

Introduction

Authorization policies play a prominent role in the security of computer and software systems: *given an access request to a sensitive resource, an authorization policy determines whether the request should be actually allowed or not.* This simple and intuitive description yields a surprising number of challenges, which require actions at very different levels.

First, we need a systematic way to express which resources are sensitive and who should be entitled to access them. This is a problem of policy *specification.* Typical authorization policies discriminate among different access modalities, providing a fine-grained form of access control: for instance, many configuration files of a standard Unix/Linux machine are publicly readable, but only the system administrator is allowed to change their content. Many real-world scenarios additionally require to state more sophisticated constraints on resource access, defining usage bounds based, e.g., on access counts, traffic loads, or geolocation. Defining expressive but convenient policy specification languages is an important issue, which has triggered the development of many research papers in the past [24].

Assume then given an authorization policy expressed in some formal, unambiguous specification language. How do we guarantee that the policy is correctly *enforced*? This is a completely different problem, which must necessarily take into account several low-level, domain-specific details. In a centralized system, for instance, one has to precisely identify all the entry points to each sensitive resource and define appropriate runtime checks; in a distributed system, instead, policy enforcement typically involves the exchange of cryptographic evidence over an untrusted network to achieve authentication. In general, filling the gap between a high-level policy specification and its low-level enforcement is a difficult and error-prone task.

Last but not least, we cannot overlook the problem of policy *verification.* Let us take for granted the existence of an appropriate policy specification language and an effective enforcement mechanism, how do we validate if the authorization policy we specified is “correct”? Namely, the security properties one has in mind while writing the policy may be stated at a different level than the policy itself, thus making their verification difficult. As a real world example, assume we regulate physical access to our research laboratory by requiring visitors to exhibit a badge before entering. This should intuitively prevent the malicious Mallory from entering our laboratory... as long as no badge ever comes under Mallory’s control! Hence, our concrete property of interest “Mallory does not have access to the lab” is not immediately entailed by our abstract policy “To get access to the lab, one must exhibit the badge”.

Luckily, formal methods have a long-standing successful tradition in preventing human mistakes and providing end-users with strong soundness guarantees. In the present thesis we focus on the development of provably sound static analysis techniques to tackle the problems of *verification* and *enforcement* of authorization policies, in a range of different settings. Static verification is useful to assess the robustness of an authorization policy way before adopting it to secure a software system, hence reducing the risks and the costs connected to the combination of the logging and auditing mechanisms necessary to retrofit a running authorization policy. Static enforcement, instead, is helpful to complement standard runtime authorization decisions: runtime enforcement is flexible and expressive, but static analysis can greatly improve performance and certify (existing) software as secure.

Summary of contributions

The first part of the thesis is devoted to the verification of Role-Based Access Control (RBAC) policies. RBAC is one of the most widespread security mechanisms in use today, and has been the subject of extensive research for more than a decade now. The central idea in the RBAC model is to factor the assignment of access rights into two steps, separating the distribution of permissions to system-specific roles, from the assignment of users to such roles, so as to simplify the overall access control management task [67]. While convenient for specification, RBAC policies are challenging for verification, in that system users can dynamically impersonate different roles at different times, thus making difficult to understand which access rights are actually granted to each user. In the present thesis we focus our attention on the formal verification of RBAC policies as implemented in `grsecurity` [71], an access control system developed on top of Unix/Linux systems. We carry out our analysis by developing the first formal semantics for the `grsecurity` RBAC system and refining it to an abstract semantics which allows for efficient model-checking of a number of diverse security properties. We implement our techniques into an automatic tool for policy verification (`gran`) and we detail the outcome of a security review performed on a number of existing `grsecurity` RBAC policies.

The second part of the thesis addresses the problem of statically enforcing *resource-aware* authorization policies in cryptographic protocols implementations. Namely, given a particular policy written in a fragment of intuitionistic affine logic, we aim at understanding if a specific protocol implementation correctly enforces such policy. We devise our framework starting from existing proposals based on refinement types [10] and extending them to support our sub-structural logic. As it turns out, designing a sound type discipline is particularly challenging, since trusted (well-typed) protocol code interacts with a powerful (untyped) Dolev-Yao opponent which controls the network. Such opponent may intercept, hijack and synthesise cryptographic messages, and mount replay attacks targeted at fooling well-typed principals into violating the resource-conscious nature of our authorization policies. On the technical side, our refinement type system does not include affine types, but it can still recover the expressiveness they would provide, by taking advantage of the underlying affine logic for authorization and our novel notion of *exponential serialization*. We show the effectiveness of our approach on two example protocols.

The third and last part of the thesis focuses on the static enforcement of an expected access control policy which is not correctly implemented by the Android platform. In fact, it is well-known that the Android permission system allows unprivileged applications to get access to sensitive resources through a transitive permissions usage enabled by the underlying message-passing system [29]. Intuitively, this violates the following (informal) authorization policy: "Access to a resource protected by a permission `P` should be allowed only to applications owning the permission `P`". We target the problem of programmatically preventing such *privilege escalation* attacks inside λ-`Perms`, a simple formal calculus which captures the essential ingredients of the Android permission and communication system, and we propose a sound security type system which statically enforces our notion of safety, despite the best efforts of an unprivileged opponent. We find our static approach particularly appealing for the Android platform, since it does not require patching the operating system to introduce additional runtime checks. We briefly discuss the design of a prototype implementation of `Lintent`, a type-checker for Android applications drawing on our formal framework. The implementation is completely due to the work of Alvise Spanò and its details can be found in the author's thesis [70].

Structure of the thesis

The thesis is structured in three parts:

- Chapter 1 discusses the formal framework for the verification of `grsecurity` RBAC policies and the related results;
- Chapter 2 presents the refinement type system for RCF. A complete soundness proof is detailed in Chapter 3;
- Chapter 4 discusses the static analysis technique aimed at preventing privilege escalation attacks on Android. Its soundness is proved in Chapter 5.

Chapter 1

Formal Verification of grsecurity RBAC Policies

1.1 Introduction

Role-Based Access Control (RBAC) provides a convenient, widely deployed security mechanism. The central idea in the RBAC model is to factor the assignment of access rights into two steps, separating the distribution of permissions to system-specific roles, from the assignment of users to such roles, so as to simplify the overall access control management task [67]. In fact, many organizations present a small and fixed set of roles with a well-understood semantics, while the effective identity of their members, and their assignments to the existing roles, are subject to frequent changes due to, e.g., hirings, firings, or promotions. The RBAC model thus supports the idea of shifting most of the effort related to policy specification inside the definitions of the roles, which are largely invariant over time.

Most of the research work on RBAC (see, e.g., [68, 7, 47]) has focused on policy *verification*, a problem of critical importance for system administrators, and a challenging one due to the complexity of the policies to be verified and to the state changes that arise in their management. State-change is not specific to RBAC: traditional access control frameworks such as those studied in the seminal work of [53] include rules to affect the structure of the access control matrix, defining the permissions granted to subjects on objects. Modern administrative RBAC (ARBAC) systems present similar features by providing system administrators with expressive languages for manipulating RBAC policies by re-assigning users to roles and/or modifying the assignment of permissions to roles [67].

Policy verification in access control systems has traditionally been stated as a safety question, answered by means of a reachability analysis: for instance, user-role reachability in ARBAC systems formalizes the problem of determining whether, given an initial policy state, a target user and a role, there exists a sequence of state changes leading to a state in which the target user is impersonating that role. As it turns out, this kind of analysis is challenging, as the procedural nature of state change languages often creates subtle, undesired effects that are hard to anticipate without the aid of a tool for analysis.

Model checking [28] has emerged as a promising technique for automated policy verification [47, 83, 56]. The idea is exactly as in program verification, with the set of state-change rules playing the role of the program to be tested, and the reachability question as the property of interest: to counter the state explosion that often affect the analysis, making it unscalable to the point of making the problem intractable [56], researchers have advocated the usage of abstraction techniques [55].

In the present chapter we continue along this line of research, focusing our attention on the formal verification of `grsecurity` [71], an access control system developed on top of Unix/Linux systems. `grsecurity` is deployed as a patch to the OS kernel that installs a reference monitor to mediate any access to the underlying OS resources; it supports the definition and dynamic enforcement of fine-grained access control policies to let users operate on objects (resources) via the subjects (executable files) provided by the underlying file system. Users are organized in roles, which are in turn structured as to identify a subset of privileged roles with higher capabilities on the system resources, and administrative control of the access control policies.

The verification problem in `grsecurity` presents much of the complexity of Administrative RBAC systems, due to the presence of policy state changes: these may arise either from explicit administrative actions for manipulating users and roles, as well as from the interaction between `grsecurity`'s access control and the facilities provided by the underlying operating system for setting user ids, hence dynamically changing users and associated roles by executing binaries operating in *setuid* mode [1]. This dependency from state changes on the executable binaries of the underlying file system further complicates the model checking problem, as it causes the size of the search space to grow unbounded in the number of states and transitions.

We tackle the problem by resorting to an *abstraction* technique, by which the behavior resulting from the unbounded set of subjects available in the underlying file system is captured by the finite number of subjects that are listed in the security policy, which represents the input of the model checker. We prove the abstraction sound and complete, and employ it to carry out a reachability analysis on RBAC policies target at unveiling (potential) security leaks, leading to unintended accesses to sensitive resources.

Contributions In this chapter, we develop a formal semantics for the `grsecurity` RBAC system, based on a labeled transition system; besides providing the fundamental building block for our analysis, the LTS semantics has proved interesting in itself, as it made it possible to understand the subtleties of `grsecurity`'s RBAC rules, and to unveil a flaw arising from the interplay between the access control systems supported by Linux and `grsecurity`. As we discuss in Section 1.3.4, this flaw makes it possible to unexpectedly bypass the imposed `grsecurity` capability restrictions when executing a setuid/setgid binary [72].

We then introduce an abstract semantics which provides a bounded, yet sound and complete, representation of the dynamic evolution of the `grsecurity` policy states arising in the formal semantics; based on that, we develop a framework for reachability analysis aimed at detecting access control leaks in any given policy.

We implement our framework in `gran`, a tool for the automatic analysis of `grsecurity` policies: the tool takes as input an RBAC policy, a user u, a set of initial states for u (associated with the possible subjects that may impersonate u) and a target file/object o, and checks whether there is a path of state changes leading to a state that grants u access to o. We provide a report of experiments we conducted with the analysis of policies in use on existing, commercial servers running `grsecurity` to implement their RBAC systems.

Structure of the chapter Section 1.2 reviews the basic concepts and notions behind `grsecurity`; Section 1.3 presents our formal semantics of the `grsecurity` RBAC system; Section 1.4 describes the abstraction for the verification of `grsecurity` policies, and shows its formal correspondence to the previous semantics; Section 1.5 describes `gran` (`grsecurity` analyzer), a tool that automatically looks for security leaks in real `grsecurity` policies; Section 1.6 illustrates `gran` at work on some case studies; Section 1.7 discusses related work.

1.2 Background on grsecurity

`grsecurity` is a patch for the Linux kernel focused on security at the operating system level. It provides many different features on latest stable kernels, implementing a "detection, prevention, and containment" model [71]. In addition to the role-based access control (RBAC) system, which is the focus of this chapter, `grsecurity` offers protection mechanisms against privilege escalation, malicious code execution and memory corruption; it also implements an advanced auditing system. `grsecurity` is typically adopted by hosting companies to harden web servers and systems providing services to locally logged users [3].

1.2.1 grsecurity RBAC

`grsecurity` complements the standard discretionary access control (DAC) mechanism provided by Linux with a form of mandatory RBAC, providing an additional layer of protection. From now on, we identify `grsecurity` with its RBAC system.

The specification of the access control requirements is provided by a *policy*, whose structure is described in Section 1.2.2. The policy defines the available roles, which can be of four different types. *User* roles are an abstraction of standard users in Linux systems, i.e., they provide a hook to extend the traditional DAC permission system with more sophisticated mechanisms, available only in `grsecurity`. *Group* roles provide a similar device for actual groups of the system. *Special* roles, instead, are not directly associated to traditional users and groups, and they are intended to provide extra privileges to normal accounts. A *default* role applies when no user, group, or special role can be granted. The mechanism of role assignment is discussed in Section 1.3.3.

1.2.2 RBAC policies

A policy defines the permissions given to each role for the different *objects* stored in the file system. A further level of granularity is introduced through the standard notion of *subject*, i.e., an abstraction of a process. Namely, permissions are not directly assigned to roles, since this would lead to a very coarse form of access control; rather, permissions are defined for pairs of the form role-subject. For instance, user `alice` could be granted read access to the object `/var` only through the subject `/bin/ls`.

```
role alice u {
role_transitions professor
subject /  {
        /
        /bin                    x
        /boot                   h
        /dev                    h
        /dev/null               w
        /dev/pts                rw
        /dev/tty                rw
        /etc                    r
}

subject /bin/su {
user_transition_allow root
group_transition_allow root
        /                       h
        /bin                    h
        /bin/su                 x
        /dev/log                rw
}
```

Table 1.1: A snippet from a `grsecurity` policy

Table 1.1 presents a snippet from a `grsecurity` policy. Even though it does not show all the features provided by `grsecurity` RBAC, it allows us to introduce the most important elements considered in our formalization. The policy defines a user role (flag 'u') `alice`, which is allowed to impersonate the special role `professor`. Transitions to specific users and groups of the underlying Linux system can be allowed or forbidden at the subject level, e.g., by the `user_transition_allow` attribute.

Permissions are specified in terms of access modalities for the objects in the policy. In this case, any process executed by `alice` is assigned the permissions defined for subject "/", except for process `/bin/su` which specifies its own set of modalities. In general, any process is accredited a set of access rights for any object in the system, according to a hierarchical matching mechanism. For instance, subject `/bin/su` inherits the rights on `/etc` by the less specific subject "/", while it overrides the permissions for `/bin` with its own. Similarly, accesses to object `/dev/log` by subject "/" are resolved in terms of the modalities listed for the less specific object `/dev`. Complete details on this mechanism are provided in Section 1.3.2.

The modalities we consider mirror standard Linux permissions for reading '`r`', writing '`w`' and executing '`x`', plus a hiding mode '`h`'. Subjects are completely unaware of the presence of any hidden object, e.g., the process `/bin/ls` does not even list the directory `/boot` when it is launched by `alice`. We ignore other available modalities, which are either irrelevant for our setting (e.g., '`p`' for ptrace rejection) or identifiable with one of the previous modalities (e.g., '`a`' for appending).

1.2.3 User and group identifiers

Before digging into the internals of `grsecurity`, we need to briefly review how users and groups are identified in Linux systems. At the kernel level, users and groups are not distinguished by names, but by numbers. We refer to these numbers as user identifiers (UIDs) and group identifiers (GIDs) respectively. When a process is started, Linux assigns it a pair of identifiers, set to the UID of the invoking user. These identifiers are called the *effective* UID and the *real* UID of the process, respectively. The effective UID determines the privileges granted to the process and is employed, e.g., for standard DAC enforcement; the real UID, instead, affects the permissions for sending signals.

This apparently simple mechanism is complicated by an important subtlety related to the execution of particular binaries in the file system. Namely, any file f may be granted the "setuid" permission, with the following effect: when f is executed, the effective UID of the process is set to the UID of the *owner* of f, irrespective of the UID of the invoking user; the real UID, instead, is set to the UID of the caller. This allows for temporary acquisition of additional privileges to perform specific tasks.

We conclude by pointing out that changing to a particular UID is considered a sensitive operation in Linux systems and requires the process to possess the *capability* `CAP_SETUID`. Capabilities provide finer-grained distribution of privileges among processes since Linux 2.2. Remarkably, capabilities are bypassed when a "setuid" binary is executed, i.e., a process spawned by a "setuid" binary is *always* allowed to set its effective UID to the UID of the owner. All the previous discussion applies similarly to GIDs.

1.3 A formal semantics for grsecurity

We propose a formal semantics for `grsecurity` in terms of a labelled transition system. We write $f : A \to B$ when f is a total function from A to B, while we write $f : A \mapsto B$ when f is partial. We let $f(a) \downarrow$ denote that f is defined on a. Let $f : A_1 \times \ldots \times A_n \mapsto B$ and let a_i range over A_i, for any $k \leq n$ we stipulate $f(a_1, \ldots, a_k) \downarrow$ if and only if $\exists a_{k+1}, \ldots, \exists a_n : f(a_1, \ldots, a_n) \downarrow$. Finally, we let $\mathcal{P}(A)$ stand for the power set of A.

1.3.1 Policies

We presuppose denumerable sets U of users and G of groups, ranged over by u and g respectively. We also let T denote the set of role types $\{\mathtt{u}, \mathtt{g}, \mathtt{s}\}$ ranged over by t; C denote the set of capabilities $\{\mathtt{set\_uid}, \mathtt{set\_gid}\}$ and M the set of access modalities $\{\mathtt{r}, \mathtt{w}, \mathtt{x}, \mathtt{h}\}$. A policy P is a 8-tuple:

$$P = (R, S, O, \mathit{perms}, \mathit{caps}, \mathit{role\_trans}, \mathit{usr\_trans}, \mathit{grp\_trans}),$$

where:

- R is a set of roles, ranged over by r. We let R_t denote the set of roles of type t and we assume that R_t and $R_{t'}$ are disjoint whenever $t \neq t'$;

- S is a set of subjects, ranged over by s, and O is a set of objects, ranged over by o. Both subjects and objects are pathnames, as we discuss below;

- $\mathit{perms} : R \times S \times O \mapsto \mathcal{P}(M)$ defines the permissions granted by the policy. Namely, if $m \in \mathit{perms}(r, s, o)$, then subject s running on behalf of role r has permission m on object o;

- $\mathit{caps} : R \times S \mapsto \mathcal{P}(C)$ determines the capabilities allowed by the policy, i.e., if $c \in \mathit{caps}(r, s)$, then subject s running on behalf of role r can acquire capability c;

- $\mathit{role\_trans} : R \to \mathcal{P}(R_{\mathtt{s}})$ defines which special roles can be impersonated by a given role;

- $\mathit{usr\_trans} : R \times S \mapsto \mathcal{P}(U)$ defines which user identities can be assumed by a subject running on behalf of a given role;

- $\mathit{grp\_trans} : R \times S \mapsto \mathcal{P}(G)$ defines which group identities can be assumed by a subject running on behalf of a given role.

We require a number of well-formedness constraints on policies which formalize a corresponding set of syntactic checks performed by `grsecurity`. (Recall that we write $\mathit{perms}(r,s)\downarrow$ to denote $\exists o \in O : \mathit{perms}(r,s,o)\downarrow$.)

1. $\forall r : \mathit{perms}(r, /)\downarrow$, i.e., all roles define at least the subject "/";
2. $\forall r, \forall s : (\mathit{perms}(r,s)\downarrow \Rightarrow \mathit{perms}(r,s,/)\downarrow)$, i.e., every subject in every role defines at least the object "/";
3. there exists a default role "$-$" such that $\forall t : - \notin R_t$.

Throughout the chapter, most definitions (notably, the semantic rules in Tables 1.2 and 1.3) and notation are to be understood as parametric with respect to a given policy. To ease readability, we do not make such dependency explicit, and just assume P as the underlying policy instead.

1.3.2 Pathnames and matching

Subjects and objects are collectively represented within `grsecurity` policies as pathnames, and these, in turn, are defined as sequences of "/"-separated names (or wildcards) as customary in Unix systems. For ease of presentation, we henceforth disregard wildcards and assume the following simplified structure of pathnames (that always presupposes a trailing "/"). Let n be a non-empty string not including "/", and let "$\cdot$" note string concatenation. Pathnames are defined by the following productions:

$$p ::= / \mid / \cdot n \cdot p$$

Pathnames are ordered according to the standard prefix order, so that p is smaller (more specific) than, or equal to, p' whenever p' is a prefix of p. Formally, the ordering relation $\sqsubseteq$ is the smallest relation closed under the following rules:

$$\text{(P-Top)}\quad p \sqsubseteq / \qquad\qquad \text{(P-Path)}\quad \frac{p \sqsubseteq p'}{/ \cdot n \cdot p \sqsubseteq / \cdot n \cdot p'}$$

Clearly, $\sqsubseteq$ is a partial order: this ordering is paramount in `grsecurity`, as it constitutes the basic building block underlying the mechanisms for associating subjects to processes, and for checking access rights on objects. Specifically, when a process spawned by the execution of a file f running on behalf of a role r tries to access a file f', `grsecurity` matches f against the most specific subject s defined in role r such that $f \sqsubseteq s$. Similarly, f' is matched against the most specific object o, defined in subject s of role r, such that $f' \sqsubseteq o$. The permissions of o are then retrieved to evaluate whether the process can be granted access to f'. For instance, according to the policy in Table 1.1, process `/bin/cat` is granted read access to `/etc/fstab`, since `/bin/cat` matches the subject "/" and `/etc/fstab` matches the object `/etc` defined there.

We formalize the matching relation as follows. For any set A of path names, we let $min(A)$ denote the minimum element of A according to the ordering $\sqsubseteq$, whenever such an element exists. Given a pathname p, we define the *matching subject* for p in role r as:

$$match\_subj(p, r) = min(\{s \mid p \sqsubseteq s \wedge perms(r, s)\downarrow\}).$$

Analogously, we define the *matching object* for p in role r under subject s as:

$$match\_obj(p, r, s) = min(\{o \mid p \sqsubseteq o \wedge perms(r, s, o)\downarrow\}).$$

Proposition 1.3.1 below and the assumption of well-formedness of the policy imply that $match\_subj(p, r)$ is always defined; instead, $match\_obj(p, r, s)$ is defined only if $perms(r, s)\downarrow$.

Proposition 1.3.1 (Chain Property). *If $p \sqsubseteq p'$ and $p \sqsubseteq p''$, then $p' \sqsubseteq p''$ or $p'' \sqsubseteq p'$.*

Proof. By induction on the sum of the depths of the derivations of $p \sqsubseteq p'$. Base case is $p \sqsubseteq / = p'$ which by (P-Top) implies $p'' \sqsubseteq p'$. Inductive case is when $p \sqsubseteq p'$ since $p = / \cdot n \cdot \hat{p}$ and $p' = / \cdot n \cdot \hat{p}'$ with $\hat{p} \sqsubseteq \hat{p}'$. Now, if $p'' = /$ we trivially have $p' \sqsubseteq p''$. Otherwise, since $p \sqsubseteq p''$ by (P-Path) it must be $p'' = / \cdot n \cdot \hat{p}''$ with $\hat{p} \sqsubseteq \hat{p}''$. By induction we have $\hat{p}' \sqsubseteq \hat{p}''$ or $\hat{p}'' \sqsubseteq \hat{p}'$ that, by applying (P-Path), gives the thesis. $\square$

1.3.3 Role assignment

Each process in `grsecurity` has a role and a subject attached to it. The assignment of the subject to the process is performed by matching the name of the running file against the list of subjects of the current role, as discussed in Section 1.3.2. Roles, instead, are assigned according to the hierarchy “special - user - group - default”. Special roles are granted through authentication to the `gradm` utility and are intended to provide extra privileges to normal user accounts: as such, they have the highest priority. User roles, instead, are applied when a process either is executed by a user with a particular UID or changes to that UID. This is possible, since the name of every user role must match up with the name of an actual user in the system, i.e., there exists a bijective partial mapping from UIDs to user roles. It is worth noticing that only the *real* UID of the process is considered for role assignment. Group roles behave similarly to user roles, but they are applied to a given process only if no user role is associated to the process UID. The default role is chosen when no other role can be given.

A further remark is in order for role assignment: even though user roles are assigned by just looking at the real UID of the process, the presence of “setuid” binaries must be considered with care. We recall that a process spawned by a “setuid” binary sets its effective UID to the UID of the owner; however, *even unprivileged* (i.e., without the capability `CAP_SETUID`) processes can always set their real UID to their effective UID [1]. Binaries with the "setuid" permission set may then come into play during the role assignment process. As usual, similar considerations apply for "setgid" files.

1.3.4 Semantics

We assume an underlying file system, i.e., a subset of a denumerable set of pathnames F, ranged over by f. (For the sake of simplicity, we do not enforce any well-formedness condition on this set, since it is not strictly needed by our semantics.) Let r_t range over $R_t \cup \{-\}$, a *state* is a 4-tuple $\sigma = \langle r_{\mathtt{s}}, u, g, f \rangle$ describing a process spawned by the execution of file f. The process may be impersonating a special role (when $r_{\mathtt{s}} \neq -$) and is running with real UID set to u and real GID set to g. We identify UIDs and GIDs with elements from a subset of U and from a subset of G, respectively. The role associated to σ is determined by the first three components of the tuple, according to the following function *role*:

$$\mathit{role}(r_{\mathtt{s}}, u, g) = \begin{cases} r_{\mathtt{s}} & \text{if } r_{\mathtt{s}} \in R_{\mathtt{s}} \\ u & \text{if } r_{\mathtt{s}} \notin R_{\mathtt{s}}, u \in R_{\mathtt{u}} \\ g & \text{if } r_{\mathtt{s}} \notin R_{\mathtt{s}}, u \notin R_{\mathtt{u}}, g \in R_{\mathtt{g}} \\ - & \text{otherwise} \end{cases}$$

The function formalizes the role assignment process, according to the hierarchy "special - user - group - default" discussed in Section 1.3.3.

Attacker Model Our semantics tracks all role transitions and subject changes allowed to a given process. The semantics depends on an underlying Linux system hosting `grsecurity`, characterized by a set of users, a set of groups and a file system, as it is apparent by the format of the states. However, we do not explicitly model any change to the previous sets and we just assume them to be denumerable; we can imagine to pick different sets after each transition, to account for the evolution of the system as a result of background operations. Intuitively, we consider a worst-case scenario, in which any possible action not conflicting with the RBAC policy is eventually performed by the process. Of course, the resulting LTS has an infinite number of states and transitions: this problem will be tackled in Section 1.4, where we will propose an abstract, finite-state semantics, specifically designed for automated security analysis.

Transitions The transition rules are presented in Table 1.2.

Rule (SETR) accounts for login operations to special roles: such transitions must be allowed by the *role_trans* function. When $r'_{\mathtt{s}} = -$, the rule models a logout from a special role, which is always permitted. Rule (SETU) describes a change of the process UID, which must be allowed by the *usr_trans* function; moreover,

(SetR)

$$\frac{\hat{r} = \mathit{role}(r_{\mathtt{s}}, u, g) \qquad r'_{\mathtt{s}} \in \mathit{role\_trans}(\hat{r}) \cup \{-\}}{\langle r_{\mathtt{s}}, u, g, f\rangle \xrightarrow{\text{set\_role}(r'_{\mathtt{s}})} \langle r'_{\mathtt{s}}, u, g, f\rangle}$$

(SetU)

$$\frac{\begin{array}{cc}\hat{r} = \mathit{role}(r_{\mathtt{s}}, u, g) & s = \mathit{match\_subj}(f, \hat{r}) \\ u' \in \mathit{usr\_trans}(\hat{r}, s) & \mathtt{set\_uid} \in \mathit{caps}(\hat{r}, s)\end{array}}{\langle r_{\mathtt{s}}, u, g, f\rangle \xrightarrow{\text{set\_UID}(u')} \langle r_{\mathtt{s}}, u', g, f\rangle}$$

(SetG)

$$\frac{\begin{array}{cc}\hat{r} = \mathit{role}(r_{\mathtt{s}}, u, g) & s = \mathit{match\_subj}(f, \hat{r}) \\ g' \in \mathit{grp\_trans}(\hat{r}, s) & \mathtt{set\_gid} \in \mathit{caps}(\hat{r}, s)\end{array}}{\langle r_{\mathtt{s}}, u, g, f\rangle \xrightarrow{\text{set\_GID}(g')} \langle r_{\mathtt{s}}, u, g', f\rangle}$$

(Exec)

$$\frac{\begin{array}{cc} \multicolumn{2}{c}{\hat{r} = \mathit{role}(r_{\mathtt{s}}, u, g)} \\ s = \mathit{match\_subj}(f, \hat{r}) & o = \mathit{match\_obj}(f', \hat{r}, s) \\ \mathtt{x} \in \mathit{perms}(\hat{r}, s, o) & \mathtt{h} \notin \mathit{perms}(\hat{r}, s, o) \\ u' \in \mathit{usr\_trans}(\hat{r}, s) \cup \{u\} & g' \in \mathit{grp\_trans}(\hat{r}, s) \cup \{g\}\end{array}}{\langle r_{\mathtt{s}}, u, g, f\rangle \xrightarrow{\text{exec}(f')} \langle r_{\mathtt{s}}, u', g', f'\rangle}$$

Table 1.2: Semantics of `grsecurity`

the process must possess the capability `set_uid`, as we discussed in Section 1.2.3. Notice that s is the matching subject for file f in the role $\hat{r}$ associated to the current state. Rule (SetG) details a similar behavior for changing the process GID. Finally, rule (Exec) accounts for the execution of files and is the most interesting rule. The invoked file must indeed be executable and it must not be hidden, since hidden files are not visible to unauthorized processes. The execution of the file may lead to a role change, as we explained in Section 1.3.3. Since we do not model which "setuid" and "setgid" binaries are actually present in the file system and we do not explicitly keep track of changes to file permissions, we simply assume that the execution of the file may trigger any user or group transition allowed by the policy for the current state. Of course, we also consider the possibility that the execution does not alter the identifiers of the process.

This subtle behavior when executing setuid/setgid programs was unknown before we started our formalization. In Section 1.6, we will illustrate that it is potentially harmful for security. This has also been reported to the main developer of `grsecurity`, who confirmed our findings. A fix has already been implemented in the latest stable release of `grsecurity` [72]. The solution consists in requiring the capabilities `CAP_SETUID`/`CAP_SETGID` to perform role transitions, even upon execution of setuid/setgid binaries.

1.4 Verification of grsecurity policies

While suitable for describing the operational behavior of `grsecurity`, the semantics presented in Section 1.3 is not amenable for security verification, as we discuss below. We thus propose a different semantics, designed for security analysis, which is an abstraction of the previous one, while being suitable to be model-checked. We also outline some properties of `grsecurity` policies which we consider interesting to verify and we formalize them in our framework.

1.4.1 An abstract semantics for grsecurity

The main problem with the presented semantics is that it hinges on many elements specific to the underlying Linux system hosting `grsecurity`, i.e., users, groups and files. Remarkably, all these elements are inherently dynamic, so any changes to them must be accounted for in the semantics to get a sound tool for security analysis. As a result, the corresponding LTS has infinite states and transitions, making security verification difficult to perform, inaccurate, or even infeasible. We thus design a simple *abstract* semantics for `grsecurity`, depending only on the content of the policy, which can be reasonably assumed to be static. If the policy happens to change during the lifetime of the hosting system, we simply consider a different LTS and we perform again any relevant analysis.

We start from some simple observations. First, we note that users and groups are immaterial to `grsecurity`, as only the role assigned to a process is relevant for access control. Second, we observe that also the actual content of the file system is somewhat disposable, since all granted permissions are determined by finding out a matching subject or object. We thus define an abstract state as a 4-tuple $\sigma_a = \langle r_{\mathtt{s}}, r_{\mathtt{u}}, r_{\mathtt{g}}, s \rangle$ describing a process spawned by the execution of some file $f \sqsubseteq s$. The role assigned to the process is again determined by the first three components of the tuple and can be retrieved by overloading the type of the function *role* defined previously.

We first abstract from impersonation of user identities. The intuition here is that, in general, only a subset of the users has an associated user role, according to the definition of the policy, and all other users can be identified by `grsecurity` to the special identity "$-$". We thus define the abstraction of a user u, denoted by $[\![u]\!]$, as follows:

$$[\![u]\!] = \begin{cases} u & \text{if } u \in R_{\mathtt{u}} \\ - & \text{otherwise} \end{cases}$$

We define the abstract version of the *usr_trans* function, denoted by $[\![usr\_trans]\!]$, as the partial function with the same domain of *usr_ trans* such that for, every r and s, we have:

$$[\![usr\_trans]\!](r, s) = \{[\![u]\!] \mid u \in usr\_trans(r, s)\}$$

In other words, transitions to users with no associated user role are collapsed to transitions to the special identity "–". We introduce analogous definitions also for groups and group transitions.

$$\begin{aligned} [\![g]\!] &= \begin{cases} g & \text{if } g \in R_{\mathtt{g}} \\ - & \text{otherwise} \end{cases} \\ [\![grp\_trans]\!](r, s) &= \{[\![g]\!] \mid g \in grp\_trans(r, s)\} \end{aligned}$$

We still need to address the most challenging task for the definition of the new semantics, i.e., the approximation of the behaviour of `grsecurity` upon file executions. The idea is to identify the executed file f with its matching object o: as a consequence of this abstraction, we can only find out an approximation for the subject to assign to the new process. This is done in terms of a set of possible matches, elaborating on the following observations:

- since o is the matching object for f, then f must be at least as specific as o ($f \sqsubseteq o$). Thus, we can take as an *upper bound* for the new subject the most specific subject which is no more specific than o, i.e., the subject $min(\{s' \mid o \sqsubseteq s'\})$. For instance, the execution of the file `/bin/ls`, matching the object `/bin/ls`, may lead to the impersonation of the subject `/bin` only if the more specific subject `/bin/ls` does not exist;
- since we do not know how much specific is f, every subject s' no more generic than o ($s' \sqsubseteq o$) may be a possible match. However, we can filter out all the subjects which would be associated to the execution of a more specific object o' which overrides o, i.e., we consider the set $\{s' \mid match\_obj(s', r, s) = o\}$, where r and s identify the current role and subject. For instance, the execution of a file in `/bin`, matching the object `/bin`, may lead to the impersonation of subject `/bin/ls` only if there does not exist the object `/bin/ls`. Indeed, when object `/bin/ls` exists, the execution of the file `/bin/ls` matches `/bin/ls` and not the less specific object `/bin`. Note that the file `/bin/ls` may even be non-executable, according to the policy specification for the object `/bin/ls`.

This reasoning leads to the following definition of *image* of an object o, given a role r and a subject s:

$$\begin{aligned} img(o, r, s) = {} & \{s' \mid match\_obj(s', r, s) = o\} \\ & \cup \{min(\{s' \mid o \sqsubseteq s'\})\} \end{aligned}$$

Again Proposition 1.3.1 and the well-formation of the policy imply that such a notion is always well-defined.

(A-SetR)

$$\frac{\hat{r} = \mathit{role}(r_{\mathtt{s}}, r_{\mathtt{u}}, r_{\mathtt{g}}) \qquad r'_{\mathtt{s}} \in \mathit{role\_trans}(\hat{r}) \cup \{-\}}{\langle r_{\mathtt{s}}, r_{\mathtt{u}}, r_{\mathtt{g}}, s\rangle \xrightarrow{\text{set\_spec}(r'_{\mathtt{s}})}_a \langle r'_{\mathtt{s}}, r_{\mathtt{u}}, r_{\mathtt{g}}, s\rangle}$$

(A-SetU)

$$\frac{\begin{array}{cc}\hat{r} = \mathit{role}(r_{\mathtt{s}}, r_{\mathtt{u}}, r_{\mathtt{g}}) & \hat{s} = \mathit{match\_subj}(s, \hat{r}) \\ r'_{\mathtt{u}} \in [\![\mathit{usr\_trans}]\!](\hat{r}, \hat{s}) & \mathtt{set\_uid} \in \mathit{caps}(\hat{r}, \hat{s})\end{array}}{\langle r_{\mathtt{s}}, r_{\mathtt{u}}, r_{\mathtt{g}}, s\rangle \xrightarrow{\text{set\_user}(r'_{\mathtt{u}})}_a \langle r_{\mathtt{s}}, r'_{\mathtt{u}}, r_{\mathtt{g}}, s\rangle}$$

(A-SetG)

$$\frac{\begin{array}{cc}\hat{r} = \mathit{role}(r_{\mathtt{s}}, r_{\mathtt{u}}, r_{\mathtt{g}}) & \hat{s} = \mathit{match\_subj}(s, \hat{r}) \\ r'_{\mathtt{g}} \in [\![\mathit{grp\_trans}]\!](\hat{r}, \hat{s}) & \mathtt{set\_gid} \in \mathit{caps}(\hat{r}, \hat{s})\end{array}}{\langle r_{\mathtt{s}}, r_{\mathtt{u}}, r_{\mathtt{g}}, s\rangle \xrightarrow{\text{set\_group}(r'_{\mathtt{g}})}_a \langle r_{\mathtt{s}}, r_{\mathtt{u}}, r'_{\mathtt{g}}, s\rangle}$$

(A-Exec)

$$\frac{\begin{array}{c}\hat{r} = \mathit{role}(r_{\mathtt{s}}, r_{\mathtt{u}}, r_{\mathtt{g}}) \\ \hat{s} = \mathit{match\_subj}(s, \hat{r}) \qquad \mathtt{x} \in \mathit{perms}(\hat{r}, \hat{s}, o) \\ \mathtt{h} \notin \mathit{perms}(\hat{r}, \hat{s}, o) \qquad r'_{\mathtt{u}} \in [\![\mathit{usr\_trans}]\!](\hat{r}, \hat{s}) \cup \{r_{\mathtt{u}}\} \\ r'_{\mathtt{g}} \in [\![\mathit{grp\_trans}]\!](\hat{r}, \hat{s}) \cup \{r_{\mathtt{g}}\} \qquad s' \in \mathit{img}(o, \hat{r}, \hat{s})\end{array}}{\langle r_{\mathtt{s}}, r_{\mathtt{u}}, r_{\mathtt{g}}, s\rangle \xrightarrow{\text{exec}(s')}_a \langle r_{\mathtt{s}}, r'_{\mathtt{u}}, r'_{\mathtt{g}}, s'\rangle}$$

Table 1.3: Abstract semantics of `grsecurity`

We finally present in Table 1.3 the reduction rules for the abstract semantics.

Rule (A-SetR) is identical to rule (SetR), while rule (A-SetU) is the counterpart of (SetU), abstracting from the users of the system. When $r'_{\mathtt{u}} = -$, the rule matches a transition to a user with no associated user role. Clearly, rule (A-SetG) behaves in the same way for group roles. Finally, rule (A-Exec) accounts for the execution of processes. Again, the choice of the new user and group role assumes a worst case scenario, in that every user and group transition which is allowed by the policy is taken into account by the rule. The new subject is drawn from the image of an executable object, according to the described approximation.

We conclude this subsection with two observations on the abstract semantics. First we note that, for any finite policy, the resulting LTS has a finite number of states and any state has a finite number of outgoing transitions, since both states and labels are built over finite sets. The LTS can then be effectively explored using standard techniques. We also underline our design choice to include states whose subject is not defined in the current role. Indeed, in our semantics we enforce an explicit match of the current subject against the subjects defined for the role. This choice leads to an increment of the size of the LTS, since we introduce a number of somewhat equivalent states; however, such a decision allows for a much more accurate security analysis, as we discuss below.

1.4.2 Correlating the two semantics

We now prove that the abstract semantics in Table 1.3 is a sound approximation of the concrete semantics in Table 1.2, in that every transition in the concrete semantics has a corresponding transition in the abstract semantics.

Formally, we abstract a file f in terms of the most specific subject which is no more specific than f itself, i.e., we let $[\![f]\!] = \mathit{min}(\{s \mid f \sqsubseteq s\})$. We can now define the abstraction of a concrete state $\sigma = \langle r_{\mathtt{s}}, u, g, f\rangle$ as the abstract state $[\![\sigma]\!] = \langle r_{\mathtt{s}}, [\![u]\!], [\![g]\!], [\![f]\!]\rangle$.

Proposition 1.4.1 (Identity Preservation). *The following equalities hold:*

(*i*) $\mathit{role}(r_{\mathtt{s}}, u, g) = \mathit{role}(r_{\mathtt{s}}, [\![u]\!], [\![g]\!])$;

(*ii*) $\mathit{match\_subj}(f, r) = \mathit{match\_subj}([\![f]\!], r)$.

Proof. For (*i*) we observe that $u = [\![u]\!]$ if $u \in R_{\mathrm{u}}$ and $g = [\![g]\!]$ if $g \in R_{\mathrm{g}}$. When, instead, $u \notin R_{\mathrm{u}}$ and $g \notin R_{\mathrm{g}}$ we have $\mathit{role}(r_{\mathtt{s}}, u, g) = \mathit{role}(r_{\mathtt{s}}, -, g) = \mathit{role}(r_{\mathtt{s}}, [\![u]\!], g)$ and $\mathit{role}(r_{\mathtt{s}}, u, g) = \mathit{role}(r_{\mathtt{s}}, u, -) = \mathit{role}(r_{\mathtt{s}}, u, [\![g]\!])$, giving the thesis. Item (*ii*) holds since $\{s \mid f \sqsubseteq s\} = \{s \mid [\![f]\!] \sqsubseteq s\}$, by definition of $[\![f]\!]$. □

Lemma 1.4.2 (Abstract Execution). *If $\mathit{match\_obj}(f, r, s) = o$, then $[\![f]\!] \in \mathit{img}(o, r, s)$.*

Proof. We first observe few, auxiliary properties. Let $p \sqsubseteq p'$, then one has:

(a) $[\![p]\!] \sqsubseteq p'$ or $p' \sqsubseteq [\![p]\!]$;

(b) $[\![p]\!] \sqsubseteq [\![p']\!]$;

(c) $\mathit{match\_obj}(p, r, s) \sqsubseteq \mathit{match\_obj}(p', r, s)$.

(a) follows directly by Proposition 1.3.1 from the observation that $p \sqsubseteq [\![p]\!]$, while (b) and (c) follow by noting that $\{\hat{p} \mid p' \sqsubseteq \hat{p}\} \subseteq \{\hat{p} \mid p \sqsubseteq \hat{p}\}$, by transitivity of $\sqsubseteq$.

We are now ready to prove the lemma. We must show that we have either $\mathit{match\_obj}([\![f]\!], r, s) = o$, or $[\![f]\!] = \mathit{min}\{s' \mid o \sqsubseteq s'\} = [\![o]\!]$. Since $\mathit{match\_obj}(f, r, s) = o$, we have $f \sqsubseteq o$. Then, by (a) we can distinguish two cases, namely $[\![f]\!] \sqsubseteq o$ or $o \sqsubseteq [\![f]\!]$:

- let $[\![f]\!] \sqsubseteq o$ and let us assume by contradiction that $\mathit{match\_obj}([\![f]\!], r, s) \neq o$. Since $\mathit{match\_obj}(f, r, s) = o$, we have $\mathit{match\_obj}(o, r, s) = o$, which implies $\mathit{match\_obj}([\![f]\!], r, s) \sqsubset o$ by (c) and assumption $\mathit{match\_obj}([\![f]\!], r, s) \neq o$. Given that $f \sqsubseteq [\![f]\!]$, we then have $\mathit{match\_obj}(f, r, s) \sqsubseteq \mathit{match\_obj}([\![f]\!], r, s)$ by (c), which implies $\mathit{match\_obj}(f, r, s) \sqsubset o$ by transitivity, giving a contradiction;

- let $o \sqsubseteq [\![f]\!]$ and let us assume by contradiction that $[\![o]\!] \sqsubset [\![f]\!]$, i.e., $[\![o]\!] \sqsubseteq [\![f]\!]$ and $[\![o]\!] \neq [\![f]\!]$. Since $f \sqsubseteq o$, we have $[\![f]\!] \sqsubseteq [\![o]\!]$ by (b), thus we have $[\![f]\!] = [\![o]\!]$ by antisymmetry, giving a contradiction.

□

Theorem 1.4.3 (Soundness)**.** *If $\sigma \xrightarrow{\alpha} \sigma'$, then there exists a label β such that $[\![\sigma]\!] \xrightarrow{\beta}_a [\![\sigma']\!]$.*

Proof. By a case analysis on the rule applied to derive $\sigma \xrightarrow{\alpha} \sigma'$. If the rule is (SetR), the conclusion follows by the first item of Proposition 1.4.1. If the rule is (SetU) or (SetG), the conclusion relies on both items of Proposition 1.4.1 that imply that $\hat{r}$ and $\hat{s}$ in the abstract semantics are the same as $\hat{r}$ and s in the concrete semantics and consequently, $[\![u']\!] \in [\![usr\_trans]\!](\hat{r}, \hat{s})$ and $[\![g]\!] \in [\![grp\_trans]\!](\hat{r}, \hat{s})$. If the rule is (Exec), we conclude again by Proposition 1.4.1, in combination with Lemma 1.4.2 which additionally implies $[\![f']\!] \in img(o, \hat{r}, \hat{s})$. □

Interestingly, our formalization enjoys also a completeness result, which states that every transition in the abstract semantics has a corresponding transition in the concrete semantics for some Linux system hosting `grsecurity`, as far as there exist at least one user and one group that do not have a corresponding role defined in the policy. This assumption bears no loss of generality for finite policies.

Lemma 1.4.4 (Concrete Execution)**.** *If $s' \in img(o, r, s)$ and $perms(r, s, o)\downarrow$, then there exists f such that $[\![f]\!] = s'$ and $match\_obj(f, r, s) = o$.*

Proof. Since $s' \in img(o, r, s)$, we can distinguish two cases. If $s' = [\![o]\!]$, we let $f = o$. Otherwise, if $s' \sqsubseteq o$ and $match\_obj(s', r, s) = o$, we let $f = s'$. □

Theorem 1.4.5 (Completeness)**.** *Consider a policy such that $\exists u, g : u \notin R_u, g \notin R_g$. If $\sigma \xrightarrow{\beta}_a \sigma'$, then there exist a label α and two concrete states $\hat{\sigma}, \hat{\sigma}'$ such that $[\![\hat{\sigma}]\!] = \sigma$, $[\![\hat{\sigma}']\!] = \sigma'$ and $\hat{\sigma} \xrightarrow{\alpha} \hat{\sigma}'$.*

Proof. By a case analysis on the rule applied to derive $\sigma \xrightarrow{\beta}_a \sigma'$. For rules (A-SetR) (A-SetU) and (A-SetG) the concrete states are the same as the abstract ones apart from the special identity "$-$" that is mapped to the u or the g that we have assumed not to belong to R_{u} and R_{g}. We rely on Lemma 1.4.4 for finding a f in concrete rule (Exec) such that $[\![f]\!]$ is the same as s' in the abstract rule (A-Exec). □

1.4.3 Security analysis

Policies in `grsecurity` are much more concise and readable than policies for other access control systems as, e.g., `SELinux` [42]. However, the plain syntactic structure of the policy does not expose a number of unintended harmful behaviors which can arise at runtime. Just to mention the simplest possible issue, the system administrator may want to prevent user `alice` from reading the files in `bob`'s home directory, but any permission set for role `alice` may be overlooked, whenever `alice` was somehow able to impersonate `bob` through a number of role transitions.

We now devise a simple formalism for verifying through our semantics if a policy is "secure". The usage of the inverted commas is intended to denote the intrinsic difficulty in answering such a question, due to the lack of an underlying *system* policy, stating the desiderata of the system administrator. General approaches to RBAC policies verification consider a declarative notion of error in terms of satisfiability of an arbitrary query [56]; more practical works, instead, are tailored around specific definitions of error, like the impersonation of undesired roles [55]. Here, we adopt the latter approach and we validate the policy with respect to some simple requirements on information access, which we consider desirable goals for realistic policies. This is a precise choice, since our research targets the development of a tool, `gran`, which should be effectively usable by system administrators. Of course, our semantics can easily fit different kind of analyses, possibly extending or generalizing those presented here.

The basic ingredient for verification consists in defining which permissions are effectively granted to a given state. Namely, we introduce two judgements $\sigma \vdash \mathsf{Read}(f)$ and $\sigma \vdash \mathsf{Write}(f)$ to denote that file f is readable (writeable) in state σ. The definition of such judgements arises as expected.

(L-Read)

$$\frac{\begin{array}{c}\hat{r} = \mathit{role}(r_{\mathtt{s}}, u, g) \\ s = \mathit{match\_subj}(f, \hat{r}) \qquad o = \mathit{match\_obj}(f', \hat{r}, s) \\ \mathtt{r} \in \mathit{perms}(\hat{r}, s, o) \qquad \mathtt{h} \notin \mathit{perms}(\hat{r}, s, o)\end{array}}{\langle r_{\mathtt{s}}, u, g, f\rangle \vdash \mathsf{Read}(f')}$$

(L-Write)

$$\frac{\begin{array}{c}\hat{r} = \mathit{role}(r_{\mathtt{s}}, u, g) \\ s = \mathit{match\_subj}(f, \hat{r}) \qquad o = \mathit{match\_obj}(f', \hat{r}, s) \\ \mathtt{w} \in \mathit{perms}(\hat{r}, s, o) \qquad \mathtt{h} \notin \mathit{perms}(\hat{r}, s, o)\end{array}}{\langle r_{\mathtt{s}}, u, g, f\rangle \vdash \mathsf{Write}(f')}$$

We assume to extend such rules to abstract states, in terms of the judgements $\sigma \vdash_a \mathsf{Read}(f)$ and $\sigma \vdash_a \mathsf{Write}(f)$.

Lemma 1.4.6 (Simple Safety). *Let $\mathcal{J}$ be either $\mathsf{Read}(f)$ or $\mathsf{Write}(f)$:*

(i) if $\sigma \vdash \mathcal{J}$, then $[\![\sigma]\!] \vdash_a \mathcal{J}$.

(ii) if $\sigma \vdash_a \mathcal{J}$, then there exists a concrete state σ' such that $[\![\sigma']\!] = \sigma$ and $\sigma' \vdash \mathcal{J}$.

Proof. This immediately follows by Proposition 1.4.1. □

All the security analyses we propose below are based on the reachability of a state with given permissions. Lemma 1.4.6, in combination with Theorem 1.4.3, guarantees that the properties can be soundly validated on the abstract semantics; in combination with Theorem 1.4.5, instead, it ensures that any security violation found in the abstract semantics has a counterpart in some Linux system. Thus, verification turns out to be decidable and can be effectively performed, as we discuss in Section 1.5. For readability, we state the analyses only for the concrete semantics.

Specification of the analyses The first analysis we propose focuses on direct accesses to files, both for reading and for writing. In particular, we want to verify if a user u can eventually get read (write) access to a given file f. While easy to specify, we believe that such property fits the needs of many system administrators, since the operational behaviour of `grsecurity` is subtler than expected. The formal description of the property we consider is reminiscent of similar specifications through temporal logics for verification like CTL and LTL [27, 64]. Namely, we define two judgements $\sigma \vdash \mathsf{ERead}(f, \sigma')$ and $\sigma \vdash \mathsf{EWrite}(f, \sigma')$ to denote that file f can *eventually* be read (written) in state σ', starting from state σ.

$$\text{(L-ERead)}\quad \frac{\sigma \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} \sigma' \qquad \sigma' \vdash \mathsf{Read}(f)}{\sigma \vdash \mathsf{ERead}(f, \sigma')}$$

The rule for $\sigma \vdash \mathsf{EWrite}(f, \sigma')$ arises as expected. We just write $\sigma \vdash \mathsf{ERead}(f)$ and $\sigma \vdash \mathsf{EWrite}(f)$ when σ' is unimportant.

Given a user u, we denote with $\mathcal{S}(u)$ the set of the *initial states* of u. Any initial state for u has the form $\langle -, u, g, f \rangle$, where g is the primary group assigned to u by the underlying Linux system and f is a possible entry point for u. For instance, `/bin/bash` may be the standard entry point for users interfacing to the system through a "bash" shell. Here, we just assume to be given a set of initial entry points for any user and we defer the discussion on the definition of such sets to Section 1.5. Note also that for initial states we are assuming that the user is not acting under any special role, since impersonation of such roles may happen only through authentication to the `gradm` utility, after a standard login operation to the Linux system.

Definition 1.4.1 (Eventual Read Access)**.** A user u can *eventually read* file f if and only if there exists $\sigma \in \mathcal{S}(u)$ such that $\sigma \vdash \mathsf{ERead}(f)$.

Eventual write access is defined accordingly.

We now build on our first analysis to specify a stronger property, inspired by the literature on information flow control [66]. We note, however, that in our setting we do not have any explicit notion of security label, so we focus on flows among different roles. Namely, if a user u_1 can read the content of file f and then write on an object o readable by u_2, then there exists a possible flow of information from u_1 to u_2 through o. This is an adaptation to our framework of the well-known "star-property" [9].

Definition 1.4.2 (Reading Flow)**.** There exists a *reading flow* on file f from user u_1 to user u_2 if and only if:

(i) there exists $\sigma \in \mathcal{S}(u_1)$ such that $\sigma \vdash \mathsf{ERead}(f, \sigma')$ and $\sigma' \vdash \mathsf{EWrite}(o)$ for some o;

(ii) there exists $\sigma'' \in \mathcal{S}(u_2)$ such that $\sigma'' \vdash \mathsf{ERead}(o)$.

Writing flows can be dually defined to address integrity issues. Again, this is just a reformulation into our setting of a standard property [14].

Definition 1.4.3 (Writing Flow)**.** There exists a *writing flow* on file f from user u_1 to user u_2 if and only if:

(i) there exists $\sigma \in \mathcal{S}(u_1)$ such that $\sigma \vdash \mathsf{EWrite}(o)$ for some o;

(ii) there exists $\sigma' \in \mathcal{S}(u_2)$ such that $\sigma' \vdash \mathsf{ERead}(o, \sigma'')$ and $\sigma'' \vdash \mathsf{EWrite}(f)$.

Note that both previous definitions ignore flows generated by multiple interacting users through a set of intermediate objects. While there is no technical difficulty in generalizing the definitions to such cases, we note that the current formulation already describes very strong properties.

The last analysis we consider accounts for a dangerous combination of permissions over the same object. Namely, if a user can acquire both permissions 'w' and 'x' on o, then o can be exploited for malicious code injection. `grsecurity` identifies this as an important problem, so it prevents the administrator from granting both said permissions for the same object; however, such a situation can arise at runtime, so we consider interesting to monitor it. We omit the formal specification of the analysis, much along the same lines of the previous proposals. We refer to Section 1.6 for details on our experiments.

1.5 Gran: a tool for policy verification

We present `gran`, a security analyser for `grsecurity` policies. The tool is written in Python and comprises around 1000 lines of code. The source code for a beta release of `gran` can be downloaded at `http://github.com/secgroup/gran`. The implementation is mostly due to Riccardo Focardi and Marco Squarcina.

Given a `grsecurity` policy, `gran` performs a pre-processing, which involves the expansion of the `include` and `replace` directives. These are just syntactic sugar, used to import fragment of other policies and to define macros, respectively. The tool then generates a model of the policy based on our formalization, i.e., it constructs a tuple $(R, S, O, \mathit{perms}, \mathit{caps}, \mathit{role\_trans}, \mathit{usr\_trans}, \mathit{grp\_trans})$.

Roles, subjects and objects are retrieved simply by parsing the policy specification. The generation of *perms* involves an unfolding of the pre-processed policy, to cope with the inheritance mechanism of `grsecurity`. We recall that, if a subject s does not specify any permission for object o, but a less specific subject defines an entry for it, then s inherits the same permissions for o. The only exception to this rule is when the subject specifies the "override" mode '`o`', which prevents this behaviour. Thus, the permissions stored in *perms* correspond to a properly unfolded version of those specified in the original policy.

Every capability is allowed by default, so for every role r and subject s we initially let $\mathit{caps}(r, s) = C$, then we remove any forbidden capability. Addition and revocation of capabilities is performed through the rules `+CAP_NAME` and `-CAP_NAME`, respectively. The overall result is order-sensitive, i.e., specifying first `+CAP_SETUID` and then `-CAP_SETUID` forbids the capability, while swapping the rules allows it. We also account for inheritance of capabilities among subjects defined in the same role.

Transitions to special roles are forbidden by default, so for every role r we initially let $\mathit{role\_trans}(r) = \emptyset$ and then we introduce in the set only the transitions explicitly allowed by the attribute `role_transitions`. Conversely, transitions to user roles are allowed by default, so we let $\mathit{usr\_trans}(r, s) = R_{\mathtt{u}} \cup \{-\}$ for any role r and subject s not providing any further specification. We recall that we abstract users with no associated user role by the distinguished identity "$-$". The attribute `user_transition_allow` can be used to restrict allowed user transitions to the ones specified. Conversely, the attribute `user_transition_deny` can be used to permit all users transitions except those listed. The two attributes cannot coexist. If subject s in role r specifies a set U of allowed user transitions, we let $\mathit{usr\_trans}(r, s) = \{[\![u]\!] \mid u \in U\}$. Conversely, if U is a set of denied user transitions, we let $\mathit{usr\_trans}(r, s) = (R_{\mathtt{u}} \setminus U) \cup \{-\}$. We apply a similar processing to construct the function *grp_ trans*.

The tool disregards features that are not modeled in this chapter, such as resource restrictions and socket policies. Domains, i.e., sets of user or group roles sharing a common set of permissions, are handled through unfolding as a set of user or group roles. Nested subjects are not supported, since the learning system of `grsecurity` does not account for them. In fact, `grsecurity` features the possibility to automatically generate a policy by inferring the right permissions from the standard usage of the system, to avoid burdening the user with the necessity of specifying all the details about access control. Since most users perform a full system learning and then tweak the generated policy around their own needs, we think nested subjects can be safely disregarded by our analysis.

After the parsing of the policy, `gran` generates all the possible states of the model and computes the set of the transitions. The tool implements all the analyses described in Section 1.4.3: the initial states and the sensible objects to consider for verification can be specified through command-line parameters. As a default choice, `gran` generates an initial state for each non-special role in the policy, assuming "/" as the subject entry point. If no target is specified for the analysis, `gran` infers a set of sensible resources by the specification provided in the configuration files of the learning system.

1.6 Case studies

We illustrate the outcome of practical experiments with `gran` and we give general considerations about possible vulnerabilities found by the tool.

1.6.1 Verification of existing policies

We asked the `grsecurity` community for policies to be verified using `gran`. Unfortunately, most system administrators are unwilling to provide their policies, since they can reveal a number of potentially harmful information about the system. However, we managed to gather a small set of real policies and we analyzed them with our tool. Due to privacy reasons, we cannot reveal any detail of such policies, so we present a properly sanitized outcome of the verification process. Our preliminary results were favorably welcome by the lead developer of `grsecurity`, who proposed us to integrate our tool in the `gradm` utility for policy management [73]. We consider this an important opportunity to continue our investigation on a larger scale, since users are for sure more comfortable to provide us the results of the validation rather than to disclose their policy.

We performed the verification of five different policies: the first and the second one from small web servers, the third one from a server running at our department, the fourth one generated by the learning system of `grsecurity`, and the fifth one from a large web server. In all cases, `gran` performed very effectively, providing the results of the analysis in less that one minute on a standard commercial machine. The output of the analysis was manually reviewed, looking for possible vulnerabilities: the process took from 10 to 30 minutes for each policy.

We start by discussing direct accesses to sensitive information. In some cases, we noticed that critical files like `/etc/shadow` were readable by untrusted users. Even though this is not a vulnerability by itself, since the underlying DAC enforced by Linux does prevent this behavior, we believe that this is a poor specification at the very least. Indeed, system critical files on a hardened server are better be protected also by a MAC policy. Interestingly, a similar warning sometimes applies also for resources which are publicly readable, according to the default settings of Linux DAC, but are considered highly sensitive by the standard configuration files of the learning system of `grsecurity`. Examples of such resources include files as `/proc/slabinfo` and `/proc/kallsyms`, whose content may be potentially exploited by an attacker. We also noticed a dangerous specification in one of the analyzed policies: subject `/etc/cron.monthly` was provided almighty access to the system.

This can have a tremendous impact on security, since cronjobs are usually executed with `root` privileges, thus mostly bypassing standard DAC. We argue that such a dangerous specification was provided for convenience, since scheduled jobs may need many different access rights and a careful assignment of permissions should feature very high granularity. Finally, we noticed that at least one of the users was not fully aware of the workings of the inheritance mechanism and, by manually tweaking the policy after the learning process, had created some unwanted cascade propagation of permissions.

We also performed some tests based on the other kinds of analyses described in Section 1.4.3. In particular, we noticed that unwanted writing accesses are much less frequent than undesired reading accesses: this is comforting and it was somehow expected, since the learning system of `grsecurity` tends to grant really few write permissions. The analysis also highlighted that usually only "physical" users, i.e., users with shell access to the system, have the opportunity to get both write and execution permissions over the same object, thus compromised services are unlikely to execute arbitrary code. Users, instead, probably need such permissions to effectively work on the system.

We think that the overall security of the analyzed `grsecurity` policies was fairly satisfying. We argue that much of the robustness, especially against undesired write accesses, comes from the sophisticated learning system of `grsecurity`, which tries to grant minimal privileges to each user. Indeed, the analyzed auto-generated policy turned out to be quite resilient to vulnerabilities; unfortunately, most administrators need to manually tweak the policy to get an usable system for their users and the overall impact of local changes may be easily overlooked. We think that our tool helps in getting the big picture on the security of the system.

1.6.2 Exploits through "setuid" binaries

The analysis presented in the previous section was performed using the "`-b`" option of `gran`, which discharges potential attacks due to the "setuid" flaw pointed out in Section 1.3. We decided to make this choice, since a fix is already going to be merged in `grsecurity`, and in our worst-case scenario almost every object of the policy turns out to be potentially vulnerable. Precisely, the amended abstract semantics assumes that no role transition can be performed upon execution. Here, we discuss the impact of the flaw we found out, by describing a realistic scenario where it can be harmfully exploited by an attacker.

One of the goal of `grsecurity` is to try to drop many of the privileges normally granted to `root`, thus limiting the impact of many known vulnerabilities; however, during the learning process, some background operations may be overlooked, leading to undesired assignment of permissions. For instance, let us assume that an administrator enables full system learning for `grsecurity` to generate his own policy. Later, during the learning phase, a scheduled cronjob process performs an access to a sensitive resource: in this case the learning system would provide `root` with liberal access rights on the resource, since it would consider it as a normal system behaviour. If the administrator does not take care in manually strengthening the policy after the learning process, “setuid” binaries can lead to unintended impersonation of a powerful `root` role, bypassing the capability system. Indeed, although the learning process tries to forbid as many capabilities as possible to user roles, such a practice does not offer the expected level of security, due to the subtle interplay between `grsecurity` and Linux.

1.6.3 Information leakage analysis

We conclude our experiments by performing an information leakage analysis on a policy we generated for testing. We agree on the common statement that compartmentalization between users is a too strict property for many realistic systems; still, it can be interesting in some highly sensitive settings [9, 14]. Our sample policy is shipped with the `gran` package and a subset of it is depicted in Table 1.4.

When we process the policy with `gran`, we find out that user `alice` is able to share some confidential information in her home directory with her accomplice `bob` through a leakage on `/tmp`. The attack is mounted on top of `cron`, which our experiments seem to identify as a subtle subject. The output of `gran` looks as follows:

```
[!!] Indirect flow found for target
     /home/alice on object /tmp
     Traces for writing:
     [1] root:U:/usr/sbin/cron
         -set_UID(alice)->
         alice:U:/usr/sbin/cron
         -exec(/usr/bin)->
         alice:U:/usr/bin/python2.7
     Traces for reading:
     [1] bob:U:/
         -exec(/bin)->
         bob:U:/bin/bash
```

We assume that `alice` can schedule her tasks through `cron`. The daemon initially runs as `root`, changes its identity to `alice` and selects for execution a Python script in `/home/alice/bin`. The subject `/usr/bin/python2.7` defined for `alice` can read `/home/alice/bin` and write on `/tmp`. `bob` cannot directly read `/tmp`, but he can execute `/bin/bash` and get read access on `/tmp`. It is worth noticing that `alice` cannot directly execute Python to get write access on `/tmp`, since her default subject “`/`” does not allow execution of files under `/usr/bin`.

Our tool is then able to identify a subtle and unintended flow, which is unlikely to be noticed by just looking at the policy. We think that `gran` can help in strengthening the system against leakage of some particularly sensitive targets.

```
role root uG
role_transitions admin
...
subject /usr/sbin/cron o {
user_transition_allow alice
group_transition_allow users
        /                        h
        /usr                     h
        /usr/sbin/cron           rx
}

role alice u
...
subject / {
        /usr/bin
}
subject /usr/sbin/cron {
        /usr/bin                 rx
}
subject /usr/bin/python2.7 o {
        /                        h
        /tmp                     rw
        /home                    r
        /home/alice/bin          r
        -CAP_ALL
}

role bob u
subject / o {
        /                        h
        /bin                     x
        -CAP_ALL
}
subject /bin/bash {
        /tmp                     rw
        /home/bob                rw
}
```

Table 1.4: A snippet of a flawed grsecurity policy

1.7 Related work

To the best of our knowledge, the present work is the first research work focusing on the verification of `grsecurity` RBAC policies. However, there exists a huge literature on the analysis of (A)RBAC policies in general, mainly targeted to the isolation of restricted classes of policies whose verification is tractable.

Sasturkar et al. [68] show that role reachability is PSPACE-complete for ARBAC and identify restrictions on the policy language to partially tame this complexity; similar results are presented by Jha et al. in later work [56]. Li and Tripunitara [57] perform a security analysis on restricted ARBAC fragments and identify a specific class of queries which can be answered efficiently. Stoller et al. [75] isolate subsets of policies of practical interest and develop algorithms to analyze them; their techniques are implemented in the RBAC-PAT tool [47], which supports also information flow analysis much in the spirit of the one provided by `gran`. Jayaraman et al. [55] propose Mohawk, a model-checker implementing an abstraction-refinement technique aimed to error finding in complex ARBAC policies.

Contrary to all these works, our framework is not targeted to the analysis of generic ARBAC policies, but of real, full-fledged `grsecurity` RBAC policies, which turn out to be amenable for efficient static verification. A formal comparison with previous work, however, might be useful to understand how possible extensions of `grsecurity` would impact on the complexity of the analysis. We leave this as future work.

Chapter 2

Enforcing Affine Authorization Policies in RCF

2.1 Introduction

Verifying the security of modern distributed applications is an important and complex challenge, which has attracted the interest of a growing research community audience over the last decade. Recent research has shown that it is possible to leverage general-purpose theorem proving techniques to develop powerful type systems for the verification of a wide range of security properties on application code, thus narrowing the gap between the formal model designed for the analysis and the actual implementation of the protocols [10, 8, 78]. The integration between type systems and theorem proving is achieved by resorting to a form of dependent types, known as *refinement* types. A refinement type $\{x : T \mid F(x)\}$ qualifies the structural information of the type T with a property specified by the logical formula F: a value M of this type is a value of type T such that $F(M)$ holds.

Authorization systems based on refinement types use the refinement formulas to express (and gain static control of) the credentials associated with the data and the cryptographic keys involved in the authorization checks. Clearly, the expressiveness of the resulting analysis hinges on the choice of the underlying logic, and indeed, several logics have been proposed for the specification and verification of security properties [24]. Other approaches, instead, have set logic parametricity as a design goal, to gain modularity and scalability of the resulting systems. Though logic parametricity is in principle a sound and wise design choice, current attempts in this direction draw primarily (if not exclusively) on classical (or intuitionistic) logical frameworks. That, in turn, is a choice that makes the resulting systems largely ineffective on large classes of resource-aware authorization policies, such as those based on consumable credentials, or predicating over access counts and/or usage bounds. The natural choice for expressing and reasoning about such classes of policies are instead *substructural* logics, such as linear and affine logic [46, 82].

On the other hand, integrating substructural logics with existing refinement type systems for distributed authorization is challenging, as one must build safeguards against the ability of an attacker to duplicate the data exchanged over the network, and correspondingly duplicate the associated credentials, thus undermining their bounded nature [19].

Contributions In this chapter, we present an *affine refinement type system* for RCF [10], a concurrent λ-calculus which can be directly mapped to a large subset of a real functional programming language like F#. The type system guarantees that well-typed programs comply with any given authorization policy expressed in affine logic, even in the presence of an active opponent.

This type system draws on the novel concept of *exponential serialization*, a general technique to protect affine formulas from the effect of duplication. This technique makes it possible to factor the authorization-relevant invariants of the analysis out of the type system, and to characterize them directly as proof obligations for the underlying affine logical system. This leads to a rather general, and modular design of our proposal, and sheds new light on the logical foundations of standard cryptographic patterns underpinning distributed authorization frameworks. Furthermore, the concept of serialization enhances the expressiveness of the type system, capturing programming patterns out of the scope of many substructural type systems.

The clean separation between typing and logical entailment has the additional advantage of enabling the formulation of an algorithmic version of our system, in which the non-deterministic proof search distinctive of substructural type systems can be dispensed with. Intuitively, we can shift all the burden related to substructural resource management into a single proof obligation to be discharged to an external theorem prover. This proof obligation can be efficiently generated from a program in a syntax-directed way: this is the key to achieve a practical implementation of our analysis technique.

We show the effectiveness of our framework on two case studies, namely the *EPMO* e-commerce protocol [52] and the *Kerberos* authentication protocol [74].

Structure of the chapter Section 2.2 overviews the challenges and the most important aspects of our theory on a simple example. Section 2.3 reviews the most important aspects of affine logic. Section 2.4 presents the meta-theory of exponential serialization. Section 2.5 reviews RCF. Section 2.6 outlines the type system and our treatment of formal cryptography. Sections 2.7-2.8 present the case studies. Section 2.9 discusses related work. All the proofs are postponed to Chapter 3.

2.2 Overview of the framework

We give an intuitive overview of our approach, based on a simple example of a distributed protocol involving a streaming service S and a client C that subscribes to the service and pays for watching a movie, chosen from a database of available contents.

2.2.1 Refinement types for verification

Verifying the protocol with a refinement type system requires to first decorate the protocol with security annotations, structured as *assumptions* and *assertions*. The former introduce logical formulas which are assumed to hold at a given point, and express the credentials available at the client side; the latter specify logical formulas which are expected to be entailed by the previously introduced assumptions, and are employed as guards for resource delivery at the server end [39, 40].

For our example, we start by assuming the authorization policy encoded by the formula: $\forall x, y.(\mathsf{Paid}(x, \$1) \Rightarrow \mathsf{Watch}(x, y))$. This is a first-order logic formula stating that each client paying one dollar can watch any movie from the database. We can then encode C and S in RCF as follows, using some standard syntactic sugar to enhance readability:

$$\begin{aligned} C &\triangleq \lambda x_C.\, \lambda x_{addS}.\, \lambda x_m.\, \lambda x_k.\, \mathsf{assume}\ \mathsf{Paid}(x_C, \$1);\\ &\quad \mathsf{let}\ x_{msg} = \mathsf{sign}\ (x_C, x_m)\ x_k\ \mathsf{in\ send}\ x_{addS}\ x_{msg}\\ S &\triangleq \lambda x_{addS}.\, \lambda x_{vk}.\, \mathsf{let}\ y_{msg} = \mathsf{recv}\ x_{addS}\ \mathsf{in}\\ &\quad \mathsf{let}\ (z_C, z_m) = \mathsf{verify}\ y_{msg}\ x_{vk}\ \mathsf{in\ assert}\ \mathsf{Watch}(z_C, z_m) \end{aligned}$$

C and S are structured as functions abstracting over the parameters defined by the protocol specification. Initially, C makes the assumption $\mathsf{Paid}(x_C, \$1)$, invokes the function sign to produce a signed request for movie x_m under her private key x_k, and sends it to S on channel x_{addS}. When S receives the message, she invokes the function verify to check the signature using the public key x_{vk}, retrieves the two components of the request z_C and z_m, and asserts the formula $\mathsf{Watch}(z_C, z_m)$. Crucially, the assertion by S is done in terms of the variables z_C and z_m occurring in her code, not of the variables x_C and x_m occurring in the code of C. The specification will be judged safe if, for all runs, the assertion made at the server side is entailed by the assumption made at the client and the underlying authorization policy.

Indeed, the specification can be proved safe, but a closer look reveals that the authorization policy is too liberal to enforce the expected access constraints. In fact, we have: $\forall x, y.(\mathsf{Paid}(x, \$1) \Rightarrow \mathsf{Watch}(x, y)), \mathsf{Paid}(C, \$1) \vdash \mathsf{Watch}(C, m) \wedge \mathsf{Watch}(C, m')$, i.e., a single payment by C allows her to arbitrarily access the movie database for unboundedly many movies. In other words, the policy does not require protection against replay attacks (to which the protocol is exposed, as can be easily seen).

2.2.2 Affine logic for specification

We can re-express the authorization policy for our example as the following affine logic formula: $!\forall x, y.(\mathsf{Paid}(x, \$1) \multimap \mathsf{Watch}(x, y))$, where the classical/intuitionistic implication ($\Rightarrow$) has been replaced by a *multiplicative* implication ($\multimap$), thus stating that each payment grants access to a *single* movie. The bang modality (!) is used to encode the authorization policy as a stable truth, which can be used arbitrarily many times in the proof.

In affine logic, given the hypotheses $!\forall x, y.(\mathsf{Paid}(x, \$1) \multimap \mathsf{Watch}(x, y))$ and $\mathsf{Paid}(C, \$1)$, one can derive $\mathsf{Watch}(C, m)$, but not the *multiplicative* conjunction $\mathsf{Watch}(C, m) \otimes \mathsf{Watch}(C, m')$, since proving the latter would require a double usage of the affine hypothesis $\mathsf{Paid}(C, \$1)$. As such, affine logic accounts for very natural specifications of resource-aware authorization policies.

2.2.3 Type-checking the example?

We move on to typing, to illustrate how refinement types are employed to provide a static account of the transfer of credentials required for authorization. In our example, this amounts to showing how to statically transfer the payment assumption $\mathsf{Paid}(x_C, \$1)$ from C to S. That assumption is needed by S to justify (type-check) her assertion $\mathsf{Watch}(z_C, z_m)$ according to the underlying authorization policy; the transfer of the assumption, in turn, is achieved by giving x_k and x_{vk} suitable types.

More precisely, assuming $x_C : T_1$ and $x_m : T_2$, the existing refinement type systems would give x_k type $\mathsf{SigKey}(x : T_1 * \{y : T_2 \mid \mathsf{Paid}(x, \$1)\})$, formalizing that x_k is a private key intended to sign a pair bearing the appropriate formula as a refinement; x_{vk} would be given the corresponding verification key type[1]. The type of x_k would thus require C to assume the formula $\mathsf{Paid}(x_C, \$1)$ upon signing, while the type of x_{vk} would allow S to retrieve the formula $\mathsf{Paid}(z_C, \$1)$ upon verification, which is enough to entail $\mathsf{Watch}(z_C, z_m)$ and make the protocol type-check.

With affine formulas, however, such solution deserves special care [19], since, if $\mathsf{Paid}(z_C, \$1)$ is extracted with no additional constraint by the type of x_{vk}, a replay attack mounted by an opponent could fool S into reusing the formula multiple times.

2.2.4 Exponential serialization to the rescue

There are various possibilities to protect the previous protocol against replay attacks. Here, we decide to run the protocol on top of a nonce-handshake, leading to the

[1]In RCF we do not have any primitive notion of cryptography and, therefore, we do not have types for cryptography in our type system. We still use this notation to simplify the presentation and we discuss the encoding of these types in Section 2.6.8.

following updated RCF encoding:

$$
\begin{aligned}
C &\triangleq \lambda x_C.\, \lambda x_{addC}.\, \lambda x_S.\, \lambda x_{addS}.\, \lambda x_m.\, \lambda x_k. \\
&\quad \mathsf{let}\ y_n = \mathsf{recv}\ x_{addC}\ \mathsf{in\ assume}\ \mathsf{Paid}(x_C, \$1); \\
&\quad \mathsf{let}\ x_{msg} = \mathsf{sign}\ (x_C, x_m, y_n)\ x_k\ \mathsf{in\ send}\ x_{addS}\ x_{msg} \\
S &\triangleq \lambda x_S.\, \lambda x_{addS}.\, \lambda x_C.\, \lambda x_{addC}.\, \lambda x_{vk}.\, \mathsf{let}\ x_n = \mathsf{mkNonce}(\,)\ \mathsf{in\ send}\ x_{addC}\ x_n; \\
&\quad \mathsf{let}\ y_{msg} = \mathsf{recv}\ x_{addS}\ \mathsf{in\ let}\ (z_C, z_m, z_n) = \mathsf{verify}\ y_{msg}\ x_{vk}\ \mathsf{in} \\
&\quad \mathsf{if}\ x_n = z_n\ \mathsf{then\ assert}\ \mathsf{Watch}(z_C, z_m) \\
\mathsf{mkNonce} &\triangleq \lambda\_ : \mathsf{unit}.\, \mathsf{let}\ x_f = \mathsf{mkFresh}()\ \mathsf{in\ assume}\ \mathsf{N}(x_f); x_f
\end{aligned}
$$

We assume to be given access to a function $\mathsf{mkFresh} : \mathsf{unit} \to \mathsf{bytes}$, which generates fresh bit-strings. The function $\mathsf{mkNonce} : \mathsf{unit} \to \{x : \mathsf{bytes} \mid \mathsf{N}(x)\}$ is a wrapper around $\mathsf{mkFresh}$, which additionally assumes the formula $\mathsf{N}(x_f)$ over the return value x_f of such a function. This new assumption is reflected by the *refined* return type of $\mathsf{mkNonce}$. Then, the typing of the key x_k may be structured as follows:

$$x_k : \mathsf{SigKey}(x : T_1 * y : T_2 * \{z : \mathsf{bytes} \mid !\,(\mathsf{N}(z) \multimap \mathsf{Paid}(x, \$1))\}),$$

to protect the affine formula $\mathsf{Paid}(x_C, \$1)$ with the *guard* $\mathsf{N}(x_n)$: if $\mathsf{N}(x_n)$ can be proved only once, also $\mathsf{Paid}(x_C, \$1)$ can be extracted only once, irrespectively of the number of signature verifications performed. Remarkably, the guarded version of $\mathsf{Paid}(x_C, \$1)$ is prefixed by the bang modality: as such, it is a stable truth and can be safely transmitted over the untrusted network, unaffected by replay attacks.

We can now discuss how the protocol is actually type-checked. When S creates the fresh nonce x_n by invoking $\mathsf{mkNonce}$, she retrieves the affine formula $\mathsf{N}(x_n)$ by the return type of the function. We refer to this predicate as a *control* formula, since it is intended to play the role of a guard in the subsequent exchange. When C gets the challenge, bound to y_n, she assumes the formula $\mathsf{Paid}(x_C, \$1)$ and signs the request (x_C, x_m, y_n). The type of the private key x_k enforces C to construct a proof of $!(\mathsf{N}(y_n) \multimap \mathsf{Paid}(x_C, \$1))$: this is indeed the most intriguing bit of our construction and it will be discussed in a moment. When S verifies the signature, she extracts the formula $!(\mathsf{N}(z_n) \multimap \mathsf{Paid}(z_C, \$1))$ proved by C, while the next conditional check allows her to know that $!(z_n = x_n)$ holds true when type-checking the "then" branch. All this information is used in combination with the authorization policy $!\forall x, y.(\mathsf{Paid}(x, \$1) \multimap \mathsf{Watch}(x, y))$ to justify the assertion $\mathsf{Watch}(z_C, z_m)$.

There is one problem left: the assumption $\mathsf{Paid}(x_C, \$1)$ available at the client C does not entail the guarded, exponential formula $!(\mathsf{N}(y_n) \multimap \mathsf{Paid}(x_C, \$1))$, which C needs to prove in order to use the key x_k to transmit her request. Our choice is to introduce a *serializer* for $\mathsf{Paid}(x_C, \$1)$ among the assumptions of C, to automatically provide for the creation of the guarded version of $\mathsf{Paid}(x_C, \$1)$. The serializer has the form:

$$!\forall x, y.(\mathsf{Paid}(x, \$1) \multimap !(\mathsf{N}(y) \multimap \mathsf{Paid}(x, \$1))),$$

that is, a universally quantified stable truth, serving for multiple communications of different predicates built over Paid. Serializers may be generated automatically for any given affine formula, and introducing them as additional assumptions is sound, in that it does not affect the set of entailed assertions, as we discuss in Section 2.4. Furthermore, serializers capture a rather general class of mechanisms for ensuring timely communications, like session keys or timestamps, which are all based on the consumption of an affine resource to assess the freshness of an exchange.

2.3 Review of affine logic

In the present chapter we focus on a simple, yet expressive, fragment of intuitionistic affine logic [82]. The syntax of formulas F is defined by the following productions:

$$\begin{array}{rcll} F & ::= & A \mid F \otimes F \mid F \multimap F \mid \forall x.F \mid\ !F \mid \mathbf{0} & \\ A & ::= & p(t_1, \ldots, t_n) \mid t = t' & p \text{ of arity } n \text{ in } \Sigma \\ t & ::= & x \mid f(t_1, \ldots, t_n) & f \text{ of arity } n \text{ in } \Sigma \end{array}$$

This is the multiplicative fragment of affine logic with conjunction ($\otimes$) and implication ($\multimap$), the universal quantifier ($\forall$), the exponential modality ($!$) to express persistent truths, logical falsity ($\mathbf{0}$) to express negation, and syntactic equality.

We presuppose an underlying signature Σ of predicate symbols ranged over by p, and function symbols, ranged over by f. The set of terms, ranged over by t, is defined by variables and function symbols as expected. We mention here that RCF terms can be encoded into the logic using the locally nameless representation of syntax with binders [30], as shown by Bengtson et al. [10]. The logical truth is written $\mathbf{1}$ and encoded as $() = ()$, where $()$ is the nullary function symbol encoding the RCF "unit" value. The negation of F, written $F^{\perp}$, is encoded as $F \multimap \mathbf{0}$, while inequality, written $t \neq t'$, is encoded as $(t = t')^{\perp}$.

The entailment relation $\Delta \vdash F$ from multiset of formulas to formulas is displayed in Table 2.1. In affine logic, rule (Weak) can be liberally applied to disregard formulas along a proof, while rule (Contr) is restricted to exponential formulas, to allow their unbounded duplication. Intuitively, the combination of the two rules enforces the following usage policy for formulas: "every formula must be used *at most* once in a proof, with the exception of exponential formulas, which can be used arbitrarily many times". This is in contrast with linear logic, where each formula must be used exactly once [46].

As informally discussed before, affine logic provides multiplicative counterparts of standard logical connectives: for instance, to prove the multiplicative conjunction $F_1 \otimes F_2$ from the hypotheses $\Delta = \Delta_1, \Delta_2$, we have to prove F_1 from Δ_1 and F_2 from Δ_2, thus each hypothesis in Δ is used either to prove F_1 or to prove F_2. Analogously, multiplicative implication $F_1 \multimap F_2$ acts as a sort of reaction, which consumes the resources needed to prove the premise F_1 to produce the conclusion F_2.

In rule (!-Right) the notation $!\Delta$ means that every formula in Δ is of the form $!F$. The rules for equality (=-Subst) and (=-Refl) are borrowed from existing sequent calculi presentations [79]. In rule (=-Subst), if the terms t and t' are not unifiable, then we consider the premise as trivially fulfilled.

$$\begin{array}{c}\text{(Ident)}\\ F \vdash F\end{array} \qquad \begin{array}{c}\text{(Weak)}\\ \dfrac{\Delta \vdash F'}{\Delta, F \vdash F'}\end{array} \qquad \begin{array}{c}\text{(Contr)}\\ \dfrac{\Delta, !F, !F \vdash F'}{\Delta, !F \vdash F'}\end{array} \qquad \begin{array}{c}\text{(Cut)}\\ \dfrac{\Delta_1 \vdash F \qquad \Delta_2, F \vdash F'}{\Delta_1, \Delta_2 \vdash F'}\end{array}$$

$$\begin{array}{c}(\otimes\text{-Left})\\ \dfrac{\Delta, F_1, F_2 \vdash F'}{\Delta, F_1 \otimes F_2 \vdash F'}\end{array} \qquad \begin{array}{c}(\otimes\text{-Right})\\ \dfrac{\Delta_1 \vdash F_1 \qquad \Delta_2 \vdash F_2}{\Delta_1, \Delta_2 \vdash F_1 \otimes F_2}\end{array} \qquad \begin{array}{c}(\multimap\text{-Left})\\ \dfrac{\Delta_1 \vdash F_1 \qquad \Delta_2, F_2 \vdash F'}{\Delta_1, F_1 \multimap F_2, \Delta_2 \vdash F'}\end{array}$$

$$\begin{array}{c}(\multimap\text{-Right})\\ \dfrac{\Delta, F_1 \vdash F_2}{\Delta \vdash F_1 \multimap F_2}\end{array} \qquad \begin{array}{c}(\forall\text{-Left})\\ \dfrac{\Delta, F\{t/x\} \vdash F'}{\Delta, \forall x.F \vdash F'}\end{array} \qquad \begin{array}{c}(\forall\text{-Right})\\ \dfrac{\Delta \vdash F \qquad x \notin \mathit{fv}(\Delta)}{\Delta \vdash \forall x.F}\end{array} \qquad \begin{array}{c}(!\text{-Left})\\ \dfrac{\Delta, F \vdash F'}{\Delta, !F \vdash F'}\end{array}$$

$$\begin{array}{c}(!\text{-Right})\\ \dfrac{!\Delta \vdash F}{!\Delta \vdash !F}\end{array} \qquad \begin{array}{c}\text{(False)}\\ \mathbf{0} \vdash F\end{array} \qquad \begin{array}{c}(=\text{-Subst})\\ \dfrac{\exists \sigma = \mathit{mgu}(t, t') \Rightarrow \Delta\sigma \vdash F\sigma}{\Delta, t = t' \vdash F}\end{array} \qquad \begin{array}{c}(=\text{-Refl})\\ \Delta \vdash t = t\end{array}$$

Table 2.1: The entailment relation $\Delta \vdash F$

2.4 Metatheory of exponential serialization

In principle, the introduction of serializers among the assumed hypotheses could alter the intended semantics of the authorization policy, due to the subtle interplay of formulas through the entailment relation in Table 2.1. Here, we isolate sufficient conditions under which exponential serialization leads to a sound protection mechanism for affine formulas.

We presuppose that the signature Σ of predicate symbols is partitioned in two sets Σ_A and Σ_C. Atomic formulas A have the form $p(t_1, \ldots, t_n)$ for some $p \in \Sigma_A$; control formulas C have the same form, though with $p \in \Sigma_C$. We identify various categories of formulas defined by the following productions:

$$\begin{array}{lll} B & ::= & A \mid B \otimes B \mid B \multimap B \mid \forall x.B \mid !B \quad \text{base formulas} \\ P & ::= & B \mid C \mid P \otimes P \quad \text{payload formulas} \\ G & ::= & C \multimap P \mid !G \quad \text{guarded formulas} \end{array}$$

Base formulas B are formulas of an authorization policy, built from atomic formulas using logical connectives. We use base formulas as security annotations in the application code. For simplicity, we dispense in this section with equalities and **0** to ensure consistency: such logical elements are used in our typed analysis, but we stipulate that they are never directly assumed in the code. (Notice that compromised principals can be modelled also without negation [10].) Payload formulas P are formulas which we want to serialize for communication over the untrusted network. Importantly, payload formulas comprise both base formulas and control formulas, which allows, e.g., for the transmission of fresh nonces to remote verifiers: this pattern is present in several authentication protocols [50]. Finally, guarded formulas G are used to model the serialized version of payload formulas, suitable for transmission. Notice also that serializers are not generated by any of the previous productions, so we let S stand for any serializer of the form $!\forall\tilde{x}.(P \multimap !(C \multimap P)$. We write $\Delta \vdash F^n$ for $\Delta \vdash F \otimes \ldots \otimes F$ (n times), with the proviso that $\Delta \vdash F^0$ stands for $\Delta \nvdash F$.

Intuitively, given a multiset of assumptions Δ, the extension of Δ with the serializers $S_1, \ldots, S_n$ is sound if Δ and its extension derive the same payload formulas. As it turns out, this is only true when Δ satisfies additional conditions, which we formalize next.

Definition 2.4.1 (Rank)**.** Let $rk : \Sigma_C \to \mathbb{N}$ be a total, injective function. Given a formula F, we define the *rank* of F with respect to rk, denoted by $rk(F)$, as follows:

$$\begin{array}{lcll} rk(p(t_1,\ldots,t_n)) & = & rk(p) & \text{if } p \in \Sigma_C \\ rk(F_1 \otimes F_2) & = & min\,\{rk(F_1), rk(F_2)\} & \\ rk(F) & = & +\infty & \text{otherwise} \end{array}$$

Definition 2.4.2 (Stratification)**.** A formula F is *stratified* with respect to a rank function rk if and only if: (i) $F = C \multimap P$ implies $rk(C) < rk(P)$; (ii) $F = P \multimap G$ implies that G is stratified; (iii) $F = \forall x.F'$ implies that F' is stratified; (iv) $F = !F'$ implies that F' is stratified. We assume F to be stratified in all the other cases. We say that a multiset of formulas Δ is stratified if and only if there exists a rank function rk such that each formula in Δ is stratified with respect to rk.

For instance, the multiset $C_1 \multimap C_2, C_2 \multimap C_3$ is stratified, given an appropriate choice of a rank function, while the multiset $C_1 \multimap C_2, C_2 \multimap C_1$ is not stratified. Stratification is required precisely to disallow such circular dependencies among control formulas and simplify the proof of our soundness result, Theorem 2.4.1 below. To prove that result, we need a further definition:

Definition 2.4.3 (Guardedness)**.** Let $\Delta = P_1, \ldots, P_m, S_1, \ldots, S_n$ be a stratified multiset of formulas. We say that Δ is *guarded* if and only if $\Delta \vdash C^k$ implies $k \leq 1$ for any control formula C.

The intuition underlying guardedness may be explained as follows. Consider a multiset Δ, a payload formula P such that $\Delta \vdash P$ and let $S = !\forall\tilde{x}.(P \multimap !(C \multimap P))$ be a serializer for P. Now, the only way that S may affect derivability is by allowing the duplication of the payload formula P via the exponential implication $!(C \multimap P)$, since the latter can be used arbitrarily often in a proof derivation. However, this effect is prevented if we are guaranteed that the control formula C guarding P is derived at most once in Δ: that is precisely what the guardedness condition ensures.

Theorem 2.4.1 (Soundness of Exponential Serialization)**.** *Let $\Delta = P_1, \ldots, P_m$. If $\Delta' = \Delta, S_1, \ldots, S_n$ is guarded and $\Delta' \vdash P$, then $\Delta \vdash P$ for all payload formulas P.*

While guardedness is convenient to use in the proof of Theorem 2.4.1, it is clearly difficult to check, since it relies on logical entailment. Fortunately, we can isolate a sufficient criterion to decide whether a multiset of formulas is guarded based on a simple syntactic check.

Proposition 2.4.2. *If $\Delta = P_1, \ldots, P_m, S_1, \ldots, S_n$ is stratified and the control formulas occurring in $P_1, \ldots, P_m$ are pairwise distinct, then Δ is guarded.*

2.5 Review of RCF

We assume collections of names (a, b, c, m, n) and variables (x, y, z). The syntax of values and expressions of RCF [10] is introduced in Table 2.2 below. The notions of free names and free variables arise as expected, according to the scope defined in the table.

$M, N ::=$		*values*
	x	variable
	$()$	unit
	(M, N)	pair
	$\lambda x. E$	function
	$h\ M$	construction ($h \in \{\mathsf{inl}, \mathsf{inr}, \mathsf{fold}\}$)
$D, E ::=$		*expressions*
	M	value
	$M\ N$	application
	$M = N$	syntactic equality
	$\mathsf{let}\ x = E\ \mathsf{in}\ E'$	let (scope of x is E')
	$\mathsf{let}\ (x, y) = M\ \mathsf{in}\ E$	pair split (scope of x, y is E)
	$\mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ E\ \mathsf{else}\ E'$	match (scope of x is E)
	$(\nu a)E$	restriction (scope of a is E)
	$E \upharpoonright E'$	fork
	$a!M$	message send
	$a?$	message receive
	$\mathsf{assume}\ F$	assumption
	$\mathsf{assert}\ F$	assertion

Table 2.2: Syntax of RCF expressions

Values include variables, unit, pairs, functions and constructions; constructors account for the creation of standard sum types and iso-recursive types. We also encode the boolean values $\mathsf{true} \triangleq \mathsf{inl}()$ and $\mathsf{false} \triangleq \mathsf{inr}()$. Expressions of RCF include standard λ-calculus constructs like values, applications, equality checks, lets, pair splits, and pattern matching, as well as primitives for concurrent, message-passing computations in the style of process algebras.

The semantics is mostly standard, so we just discuss the most peculiar constructs. Expression $(\nu a)E$ generates a globally fresh channel name a and then behaves as E. Expression $E \Rsh E'$ evaluates E and E' in parallel, and returns the result of E'. Expression $a!M$ asynchronously outputs M on channel a and returns $()$. Expression $a?$ waits until a term N is available on channel a and returns N. These message-passing expressions can be used to model the sending and receiving functions send and recv used in the code of our examples. The formal semantics of RCF expressions is defined by the reduction rules in Table 2.3.

$(\lambda x.\, E)\ N \to E\{N/x\}$	(RED FUN)
$\mathsf{let}\ (x, y) = (M, N)\ \mathsf{in}\ E \to E\{M/x\}\{N/y\}$	(RED SPLIT)
$\mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ E\ \mathsf{else}\ E' \to \begin{cases} E\{N/x\} & \text{if } M = h\ N \text{ for some } N \\ E' & \text{otherwise} \end{cases}$	(RED MATCH)
$M = N \to \begin{cases} \mathsf{true} & \text{if } M = N \\ \mathsf{false} & \text{otherwise} \end{cases}$	(RED EQ)
$a!M \Rsh a? \to M$	(RED COMM)
$\mathsf{let}\ x = M\ \mathsf{in}\ E \to E\{M/x\}$	(RED LET VAL)
$\mathsf{let}\ x = E\ \mathsf{in}\ E'' \to \mathsf{let}\ x = E'\ \mathsf{in}\ E'' \quad \text{if } E \to E'$	(RED LET)
$(\nu a)E \to (\nu a)E' \quad \text{if } E \to E'$	(RED RES)
$E \Rsh E'' \to E' \Rsh E'' \quad \text{if } E \to E'$	(RED FORK 1)
$E'' \Rsh E \to E'' \Rsh E' \quad \text{if } E \to E'$	(RED FORK 2)
$E \to E' \quad \text{if } E \Rrightarrow D, D \to D', D' \Rrightarrow E'$	(RED HEAT)

Table 2.3: Reduction semantics for RCF

The reduction semantics depends upon the heating relation $E \Rrightarrow E'$, an asymmetric version of the standard structural congruence, to perform some syntactic rearrangements of expressions and allow reductions. We write $E \equiv E'$ to denote that both $E \Rrightarrow E'$ and $E' \Rrightarrow E$. The definition of the heating relation is presented in Table 2.4, the only difference with respect to the original RCF presentation is the introduction of the rule (HEAT ASSERT ()), which simplifies our definition of safety discussed below.

We are now ready to adapt to our resource-aware setting the formal notion of *safety* defined for RCF expressions. Intuitively, an expression E is safe when, for all runs, the multiplicative conjunction of the top-level assertions is entailed by the top-level assumptions. Giving a precise definition, however, is somewhat tricky, since we need to introduce *structures*. Let e denote an *elementary* expression, e.g.,

$E \Rrightarrow E$	(Heat Refl)
$E \Rrightarrow E''$ if $E \Rrightarrow E'$ and $E' \Rrightarrow E''$	(Heat Trans)
$\mathsf{let}\ x = E\ \mathsf{in}\ E'' \Rrightarrow \mathsf{let}\ x = E'\ \mathsf{in}\ E''$ if $E \Rrightarrow E'$	(Heat Let)
$(\nu a)E \Rrightarrow (\nu a)E'$ if $E \Rrightarrow E'$	(Heat Res)
$E \Rsh E'' \Rrightarrow E' \Rsh E''$ if $E \Rrightarrow E'$	(Heat Fork 1)
$E'' \Rsh E \Rrightarrow E'' \Rsh E'$ if $E \Rrightarrow E'$	(Heat Fork 2)
$() \Rsh E \equiv E$	(Heat Fork ())
$a!M \Rrightarrow a!M \Rsh ()$	(Heat Msg ())
$\mathsf{assume}\ F \Rrightarrow \mathsf{assume}\ F \Rsh ()$	(Heat Assume ())
$\mathsf{assert}\ F \Rrightarrow \mathsf{assert}\ F \Rsh ()$	(Heat Assert ())
$E' \Rsh (\nu a)E \Rrightarrow (\nu a)(E' \Rsh E)$ if $a \notin fn(E')$	(Heat Res Fork 1)
$(\nu a)E \Rsh E' \Rrightarrow (\nu a)(E \Rsh E')$ if $a \notin fn(E')$	(Heat Res Fork 2)
$\mathsf{let}\ x = (\nu a)E\ \mathsf{in}\ E' \Rrightarrow (\nu a)(\mathsf{let}\ x = E\ \mathsf{in}\ E')$ if $a \notin fn(E')$	(Heat Res Let)
$(E \Rsh E') \Rsh E'' \equiv E \Rsh (E' \Rsh E'')$	(Heat Fork Assoc)
$(E \Rsh E') \Rsh E'' \Rrightarrow (E' \Rsh E) \Rsh E''$	(Heat Fork Comm)
$\mathsf{let}\ x = (E \Rsh E')\ \mathsf{in}\ E'' \equiv E \Rsh (\mathsf{let}\ x = E'\ \mathsf{in}\ E'')$	(Heat Fork Let)

Table 2.4: Heating relation for RCF

any expression that is not an assumption, assertion, restriction, let, fork, or send. Structures formalize the idea that a computation state has four components:

1. a series of elementary expressions e_ℓ being evaluated in parallel contexts;
2. a series of messages M_k sent on channels but not yet received;
3. a multiset of assumed formulas F_i;
4. a multiset of asserted formulas F'_j.

The definition of a structure $\mathbf{S}$ is given in Table 2.5 below. Structures are amenable for their simple notion of *static safety*, in that their syntactic form already exhibits all the necessary ingredients to state it.

$$\Pi_{i\in[1,n]}E_i \triangleq () \Rsh E_1 \Rsh \cdots \Rsh E_n$$
$$\mathcal{L}[e] ::= e \mid \mathsf{let}\ x = \mathcal{L}[e]\ \mathsf{in}\ E$$
$$\mathbf{S} ::= (\nu\widetilde{a})((\Pi_{i\in[1,m]}\mathsf{assume}\ F_i) \Rsh (\Pi_{j\in[1,n]}\mathsf{assert}\ F'_j) \Rsh (\Pi_{k\in[1,o]}c_k!M_k) \Rsh (\Pi_{\ell\in[1,p]}\mathcal{L}_\ell[e_\ell]))$$

The structure $\mathbf{S}$ is *statically safe* if and only if $F_1, \dots, F_m \vdash F'_1 \otimes \dots \otimes F'_n$.

Table 2.5: Structures and static safety

We can prove that every expression E can be transformed into a structure by heating, hence we can easily define a suitable notion of safety for any expression.

Lemma 2.5.1 (Structure). *For every expression E, there exists a structure $\mathbf{S}$ such that $E \Rrightarrow \mathbf{S}$.*

Proof. By induction on the structure of E. □

Definition 2.5.1 (Safety)**.** A closed expression E is *safe* if and only if, for all E' and $\mathbf{S}$, if $E \rightarrow^* E'$ and $E' \Rrightarrow \mathbf{S}$, then $\mathbf{S}$ is statically safe.

The real property of interest, however, is stronger than the previous one: we desire protection despite the best efforts of an active opponent. We let an *opponent* be any closed expression of RCF which does not contain any assertion. The latter is a standard restriction, since opponents containing arbitrary assertions could vacuously falsify the property we target; this does not involve any loss of generality, since we want to verify application code with respect to the annotations placed therein.

Definition 2.5.2 (Robust Safety)**.** A closed expression E is *robustly safe* if and only if, for any opponent O, the application $O\ E$ is safe.

In the previous definition, we use the standard syntactic sugar $O\ E$ for the expression $\mathsf{let}\ x = O\ \mathsf{in}\ \mathsf{let}\ y = E\ \mathsf{in}\ x\ y$.

2.6 The type system

Our refinement type system builds on previous work by Bengtson et al. [10], extending it to guarantee the correct usage of affine formulas and to enforce our revised notion of (robust) safety.

2.6.1 Types, environments, and base judgements

The syntax of types is defined in Table 2.6. Again the notions of free names and free variables arise as expected, according to the scope defined in the table.

$T, U, V ::=$		*types*
	unit	unit type
	$x : T \rightarrow U$	dependent function type (scope of x is U)
	$x : T * U$	dependent pair type (scope of x is U)
	$T + U$	sum type
	$\mu\alpha.T$	iso-recursive type
	α	type variable
	$\{x : T \mid F\}$	refinement type (scope of x is F)

Table 2.6: Syntax of types

The unit value () is given type unit. Sum types have the form $T+U$, iso-recursive types are denoted by $\mu\alpha.T$, and type variables are denoted by α. There exist various forms of dependent types: a function of type $x : T \rightarrow U$ takes as an input a value M of type T and returns a value of type $U\{M/x\}$; a pair (M, N) has type $x : T * U$ if M has type T and N has type $U\{M/x\}$; a value M has a refinement type $\{x : T \mid F\}$

if M has type T and the formula $F\{M/x\}$ holds true. We use type $\mathsf{Un} \triangleq \mathsf{unit}$ to model data that may come from, or be sent to the opponent, as it is customary for security type systems. Type $\mathsf{bool} \triangleq \mathsf{unit} + \mathsf{unit}$ is inhabited by $\mathsf{true} \triangleq \mathsf{inl}()$ and $\mathsf{false} \triangleq \mathsf{inr}()$.

Our type system comprises several typing judgements of the form $\Gamma; \Delta \vdash \mathcal{J}$, where $\Gamma; \Delta$ is a typing environment collecting all the information which can be used to derive $\mathcal{J}$. In particular, Γ contains the type bindings, while Δ comprises logical formulas that are supposed to hold at run-time. Formally, we let Γ be an ordered list of entries $\mu_1, \ldots, \mu_n$ and Δ be a multiset of affine logic formulas. Each entry μ_i in Γ denotes either a type variable (α), a kinding annotation ($\alpha :: k$), or a type binding for channels ($a \updownarrow T$) or variables ($x : T$). We let ε denote the empty list and $\emptyset$ the empty multiset. The *domain* of Γ, written $dom(\Gamma)$, is defined in Table 2.7.

$$\begin{array}{l}
dom(\alpha) = \{\alpha\} \\
dom(\alpha :: k) = \{\alpha\} \\
dom(a \updownarrow T) = \{a\} \\
dom(x : T) = \{x\} \\
dom(\mu_1, \ldots, \mu_n) = dom(\mu_1), \ldots, dom(\mu_n)
\end{array}$$

Table 2.7: Domain of Γ

We use the judgement $\Gamma; \Delta \vdash \diamond$ to denote that the typing environment $\Gamma; \Delta$ is well-formed, i.e., it satisfies some standard syntactic conditions (for instance, it does not contain duplicate type bindings for the same variable). The only remarkable point in the definition of $\Gamma; \Delta \vdash \diamond$ is that we forbid variables in Γ to be mapped to a refinement type: indeed, when extending a typing environment with a new type binding $x : T$, we use the function ψ to place the structural type information in Γ and the function *forms* to place the associated refinements in Δ. The complete definition of the judgement $\Gamma; \Delta \vdash \diamond$ is given in Table 2.8.

$$\psi(U) = \begin{cases} \psi(T) & \text{if } U = \{x : T \mid F\} \\ U & \text{otherwise} \end{cases}$$

$$forms(y : U) = \begin{cases} F\{y/x\}, forms(y : T) & \text{if } U = \{x : T \mid F\} \\ \emptyset & \text{otherwise} \end{cases}$$

(Env Empty)
$$\varepsilon; \emptyset \vdash \diamond$$

(Type Env Entry)
$$\frac{\Gamma; \Delta \vdash \diamond \qquad dom(\mu) \cap dom(\Gamma) = \emptyset \qquad \mu = x : T \Rightarrow T = \psi(T) \wedge fnfv(T) \subseteq dom(\Gamma)}{\Gamma, \mu; \Delta \vdash \diamond}$$

(Form Env Entry)
$$\frac{\Gamma; \Delta \vdash \diamond \qquad fnfv(F) \subseteq dom(\Gamma)}{\Gamma; \Delta, F \vdash \diamond}$$

Table 2.8: Well-formed environments

We write $\Gamma;\Delta \vdash T$ to note that type T is well-formed in $\Gamma;\Delta$. The standard definition of the judgement comprises only rule (Type) below.

$$\text{(Type)}\quad \frac{\Gamma;\Delta \vdash \diamond \qquad \mathit{fnfv}(T) \subseteq \mathit{dom}(\Gamma)}{\Gamma;\Delta \vdash T}$$

Finally, we use the judgement $\Gamma;\Delta \vdash F$ to denote that the formulas in Δ entail the formula F. The formal definition corresponds to rule (Derive) below.

$$\text{(Derive)}\quad \frac{\Gamma;\Delta \vdash \diamond \qquad \mathit{fnfv}(F) \subseteq \mathit{dom}(\Gamma) \qquad \Delta \vdash F}{\Gamma;\Delta \vdash F}$$

We often write $\Gamma;\Delta \vdash F_1,\ldots,F_n$ to denote that $\Gamma;\Delta \vdash F_1 \otimes \ldots \otimes F_n$, with the proviso that $\Gamma;\Delta \vdash \emptyset$ stands for $\Gamma;\Delta \vdash \mathbf{1}$.

2.6.2 Environment rewriting

We stipulate that all the type information stored in Γ can be used arbitrarily often in the derivation of any judgement of our type system, hence we dispense from affine types: in Section 2.6.7 we thoroughly discuss how we can recover expressiveness by encoding them through exponential serialization. The treatment of the formulas in Δ is subtler, since affine resources must be used at most once during type-checking: in particular, we need to split the environment Δ among subderivations to avoid the unbounded duplication of the formulas therein.

The general structure of the rules of our system then looks as follows:

$$\frac{\Gamma;\Delta_1 \vdash \mathcal{J}_1 \qquad \ldots \qquad \Gamma;\Delta_n \vdash \mathcal{J}_n \qquad \Gamma;\Delta \hookrightarrow \Gamma;\Delta_1,\ldots,\Delta_n}{\Gamma;\Delta \vdash \mathcal{J}}$$

where $\Gamma;\Delta \hookrightarrow \Gamma;\Delta'$ denotes the *environment rewriting* of $\Gamma;\Delta$ to $\Gamma;\Delta'$. Such relation is defined by the rule below:

$$\text{(Rewrite)}\quad \frac{\Delta \vdash \Delta' \qquad \Gamma;\Delta \vdash \diamond \qquad \Gamma;\Delta' \vdash \diamond}{\Gamma;\Delta \hookrightarrow \Gamma;\Delta'}$$

where we write $\Delta \vdash F_1,\ldots,F_n$ to denote that $\Delta \vdash F_1 \otimes \ldots \otimes F_n$, again with the proviso that $\Delta \vdash \emptyset$ stands for $\Delta \vdash \mathbf{1}$.

The adoption of the environment rewriting relation as an house-keeping device for the formulas of Δ greatly improves the expressiveness of the type system in a very natural way. This idea of extending to the typing environment a number of context manipulation rules from the underlying sub-structural logic was first proposed by Mandelbaum et al. [59], even though their solution is technically different from ours. Namely, the authors of [59] allow for applications of arbitrary left rules from the logic inside the typing environment, while our proposal is reminiscent of the (Cut) rule typical of sequent calculi. We find this solution simpler to present and more convenient to prove sound.

Interestingly, all the non-determinism introduced by the application of the rewriting rules and the splitting of the logical formulas among the premises of the type rules can be effectively tamed by the algorithmic type system discussed in Section 2.6.9.

2.6.3 Kinding and subtyping

Security type systems often rely on a kinding relation to discriminate whether or not messages of a specific type may be known to the attacker or received from it. The kinding judgement $\Gamma; \Delta \vdash T :: k$ denotes that type T is of kind k. We distinguish between two kinds: kind $k = \mathsf{pub}$ denotes that the inhabitants of a given type are public and may be sent to the attacker, while kind $k = \mathsf{tnt}$ denotes that the inhabitants of a given type are tainted and may come from the attacker. In the following we let $\overline{\mathsf{pub}} \triangleq \mathsf{tnt}$ and $\overline{\mathsf{tnt}} \triangleq \mathsf{pub}$.

The complete kinding relation is given in Table 2.9. Most of the rules resemble those presented in other security type systems [10, 8] and only differ in the treatment of affine formulas, which is similar to the one we employ for typing values and expressions. We postpone the discussion on this point until the next section, where it will be easier to provide an intuitive understanding.

(Kind Var)
$$\frac{\Gamma; \Delta \vdash \diamond \qquad (\alpha :: k) \in \Gamma}{\Gamma; \Delta \vdash \alpha :: k}$$

(Kind Unit)
$$\frac{\Gamma; \Delta \vdash \diamond}{\Gamma; \Delta \vdash \mathsf{unit} :: k}$$

(Kind Fun)
$$\frac{\begin{array}{c}\Gamma; !\Delta_1 \vdash T :: \overline{k} \\ \Gamma, x : \psi(T); !\Delta_2 \vdash U :: k \\ \Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2\end{array}}{\Gamma; \Delta \vdash x : T \to U :: k}$$

(Kind Pair)
$$\frac{\begin{array}{c}\Gamma; !\Delta_1 \vdash T :: k \\ \Gamma, x : \psi(T); !\Delta_2 \vdash U :: k \\ \Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2\end{array}}{\Gamma; \Delta \vdash x : T * U :: k}$$

(Kind Sum)
$$\frac{\begin{array}{c}\Gamma; !\Delta_1 \vdash T :: k \\ \Gamma; !\Delta_2 \vdash U :: k \\ \Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2\end{array}}{\Gamma; \Delta \vdash T + U :: k}$$

(Kind Rec)
$$\frac{\begin{array}{c}\Gamma, \alpha :: k; !\Delta' \vdash T :: k \\ \Gamma; \Delta \hookrightarrow \Gamma; !\Delta'\end{array}}{\Gamma; \Delta \vdash \mu\alpha. T :: k}$$

(Kind Refine Public)
$$\frac{\Gamma; \Delta \vdash \{x : T \mid F\} \qquad \Gamma; \Delta \vdash T :: \mathsf{pub}}{\Gamma; \Delta \vdash \{x : T \mid F\} :: \mathsf{pub}}$$

(Kind Refine Tainted)
$$\frac{\begin{array}{c}\Gamma; \Delta_1 \vdash \psi(T) :: \mathsf{tnt} \\ \Gamma, y : \psi(T); \Delta_2 \vdash \mathit{forms}(y : T) \\ \Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2 \qquad T \text{ refined}\end{array}}{\Gamma; \Delta \vdash T :: \mathsf{tnt}}$$

Table 2.9: Kinding

Here, we just point out some simple observations, which should hopefully guide the reader in understanding a few important aspects. For instance, a pair type is public if both its components are public and can be disclosed to the opponent: in fact, knowing a pair allows to learn both its components by splitting it. Conversely, a pair is tainted if both its components are tainted, given that, if even a single component is not tainted, then the pair cannot come from the opponent. A function type is public only if its return type is public (otherwise $\lambda x.\, M_{\text{secret}}$ could be public) and its argument type is tainted (otherwise $\lambda x.\, \mathsf{let}\ y = \mathsf{enc}(M_{\text{secret}}, x)\ \mathsf{in}\ net!y$ could be public). A refinement type is public if the structural type it refines is public, while it is tainted if the structural type it refines is tainted and the associated refinements can be proved by the current logical environment.

The subtyping judgment $\Gamma; \Delta \vdash T <: U$ expresses the fact that T is a subtype of U and, thus, values of type T can be used in place of values of type U. The complete presentation of the subtyping relation can be found in Table 2.10.

(Sub Refl)
$$\frac{\Gamma; \Delta \vdash T}{\Gamma; \Delta \vdash T <: T}$$

(Sub Pub Tnt)
$$\frac{\Gamma; \Delta_1 \vdash T :: \mathsf{pub} \qquad \Gamma; \Delta_2 \vdash U :: \mathsf{tnt} \qquad \Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2}{\Gamma; \Delta \vdash T <: U}$$

(Sub Fun)
$$\frac{\begin{array}{c}\Gamma; !\Delta_1 \vdash T' <: T \\ \Gamma, x : \psi(T'); !\Delta_2 \vdash U <: U' \\ \Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2\end{array}}{\Gamma; \Delta \vdash x : T \to U <: x : T' \to U'}$$

(Sub Pair)
$$\frac{\begin{array}{c}\Gamma; !\Delta_1 \vdash T <: T' \\ \Gamma, x : \psi(T); !\Delta_2 \vdash U <: U' \\ \Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2\end{array}}{\Gamma; \Delta \vdash x : T * U <: x : T' * U'}$$

(Sub Sum)
$$\frac{\begin{array}{c}\Gamma; !\Delta_1 \vdash T <: T' \qquad \Gamma; !\Delta_2 \vdash U <: U' \\ \Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2\end{array}}{\Gamma; \Delta \vdash T + U <: T' + U'}$$

(Sub Pos Rec)
$$\frac{\begin{array}{c}\Gamma, \alpha; !\Delta' \vdash T <: T' \\ \alpha \text{ occurs only positively in } T \text{ and } T' \\ \Gamma; \Delta \hookrightarrow \Gamma; !\Delta'\end{array}}{\Gamma; \Delta \vdash \mu\alpha.T <: \mu\alpha.T'}$$

(Sub Refine)
$$\frac{\begin{array}{c}\Gamma; \Delta_1 \vdash \psi(T) <: \psi(U) \qquad \Gamma, y : \psi(T); \Delta_2, \mathit{forms}(y : T) \vdash \mathit{forms}(y : U) \\ \Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2 \qquad T \text{ and/or } U \text{ refined}\end{array}}{\Gamma; \Delta \vdash T <: U}$$

Table 2.10: Subtyping

The subtyping judgment makes public types subtype of tainted types through rule (Sub Pub Tnt), and further describes standard subtyping relations for types sharing the same structure: for instance, pair types are covariant, while function types are contravariant in their argument and covariant in their return types. Intuitively, this means that a function can safely replace another function if it is "more liberal" in the types it accepts and "more conservative" in the types it returns. The rule for isorecursive types (Sub Pos Rec) is borrowed from [8] and it differs from the standard Amber rule proposed in the original presentation of RCF: the rule we consider here is easier to prove sound and the loss of expressiveness is very mild. We refer the interested reader to [8] for further discussion on this technical point.

The most interesting subtyping rule in Table 2.10 is (Sub Refine), which subsumes the rules (Sub Refine Left) and (Sub Refine Right) from the original presentation of RCF, which are shown below.

$$\text{(Sub Refine Left)}\quad \frac{\Gamma \vdash \{x : T \mid F\} \qquad \Gamma \vdash T <: U}{\Gamma \vdash \{x : T \mid F\} <: U} \qquad\qquad \text{(Sub Refine Right)}\quad \frac{\Gamma \vdash T <: U \qquad \Gamma, x : T \vdash F}{\Gamma \vdash T <: \{x : U \mid F\}}$$

The first rule allows to discard unneeded logical formulas and conforms to the core idea of "refinement" typing: values of type $\{x : T \mid F\}$ can be safely replaced for values of type T, since they are just values of type T with the additional information F. The second rule, instead, generalizes the substitution principle underlying subtyping to the refinement formulas: for instance, we have $\{x : T \mid x = 5\} <: \{x : T \mid x > 0\}$, since the condition $x = 5$ is stronger than the condition $x > 0$.

A natural adaptation of (Sub Refine Right) to our affine setting would be:

$$\text{(Sub Refine Wrong)}\quad \frac{\Gamma; \Delta_1 \vdash T <: U \qquad \Gamma, x : \psi(T); \Delta_2, \mathit{forms}(x : T) \vdash F \qquad \Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2}{\Gamma; \Delta \vdash T <: \{x : U \mid F\}}$$

Unfortunately, this rule is unsound, since the affine formulas of T could actually be used twice and we could prove, for instance:

$$\emptyset; \varepsilon \vdash \{x : \mathsf{Un} \mid F\} <: \{z : \{x : \mathsf{Un} \mid F\} \mid F\}.$$

This cannot happen with our new rule, since $F \nvdash F \otimes F$ in affine logic. While it is in principle possible to find out other sound counterparts of (Sub Refine Right) in our affine setting, our previous work [19] highlighted that the technical treatment of these rules is rather complicated, and we find rule (Sub Refine) much more convenient in our proofs. The previous discussion should have also clarified the reasons behind a slightly more restrictive treatment for subtyping pairs and functions with respect to the original RCF paper.

2.6.4 Typing values

The typing judgement $\Gamma; \Delta \vdash M : T$ denotes that value M is given type T under environment $\Gamma; \Delta$. The typing rules for values are given in Table 2.11.

Rule (Val Refine) is a natural adaptation to an affine setting of the standard rule for refinement types: a value M has type $\{x : T \mid F\}$ if M has type T and

$$\textsc{(Val Var)}\quad \frac{\Gamma;\Delta \vdash \diamond \qquad (x:T)\in\Gamma}{\Gamma;\Delta \vdash x : T}
\qquad
\textsc{(Val Unit)}\quad \frac{\Gamma;\Delta \vdash \diamond}{\Gamma;\Delta \vdash () : \mathsf{unit}}
\qquad
\textsc{(Val Fun)}\quad \frac{\begin{array}{c}\Gamma, x:\psi(T); !\Delta', \mathit{forms}(x:T) \vdash E : U\\ \Gamma;\Delta \hookrightarrow \Gamma; !\Delta'\end{array}}{\Gamma;\Delta \vdash \lambda x.\, E : x:T \to U}$$

$$\textsc{(Val Pair)}\quad \frac{\begin{array}{c}\Gamma; !\Delta_1 \vdash M : T \qquad \Gamma; !\Delta_2 \vdash N : U\{M/x\}\\ \Gamma;\Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2\end{array}}{\Gamma;\Delta \vdash (M,N) : x:T * U}
\qquad
\textsc{(Val Refine)}\quad \frac{\begin{array}{c}\Gamma; \Delta_1 \vdash M : T \qquad \Gamma; \Delta_2 \vdash F\{M/x\}\\ \Gamma;\Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2\end{array}}{\Gamma;\Delta \vdash M : \{x:T \mid F\}}$$

$$\textsc{(Val Inl)}\quad \frac{\begin{array}{c}\Gamma; !\Delta' \vdash M : T\\ \Gamma; !\Delta' \vdash U\\ \Gamma;\Delta \hookrightarrow \Gamma; !\Delta'\end{array}}{\Gamma;\Delta \vdash \mathsf{inl}\ M : T + U}
\qquad
\textsc{(Val Inr)}\quad \frac{\begin{array}{c}\Gamma; !\Delta' \vdash M : U\\ \Gamma; !\Delta' \vdash T\\ \Gamma;\Delta \hookrightarrow \Gamma; !\Delta'\end{array}}{\Gamma;\Delta \vdash \mathsf{inr}\ M : T + U}
\qquad
\textsc{(Val Fold)}\quad \frac{\begin{array}{c}\Gamma; !\Delta' \vdash M : T\{\mu\alpha.T/\alpha\}\\ \Gamma;\Delta \hookrightarrow \Gamma; !\Delta'\end{array}}{\Gamma;\Delta \vdash \mathsf{fold}\ M : \mu\alpha.T}$$

Table 2.11: Typing rules for values

the formula $F\{M/x\}$ holds true. Rules (Val Fun) and (Val Pair) are more interesting: recall, in fact, that our type system does not incorporate affine types, since the type information in Γ is propagated to all the premises of a typing rule. It is then crucial for soundness that both pairs and functions are type-checked in an *exponential* environment, i.e., an environment of the form $!\Delta = !F_1, \dots, !F_n$. For instance, using an affine formula F from the typing environment to give a pair (M, N) type $x : T * \{y : U \mid F\}$ would lead to an unbounded usage of F upon replicated pair splitting operations on (M, N), as we discuss in Section 2.6.7. Similar restrictions apply also to sum types and iso-recursive types.

Allowing for affine refinements, but forbidding affine types, confines the problem of resource management to the formula environment, thus simplifying the type system. In Section 2.6.7 we explain how the exponential serialization technique can be leveraged to encode affine types in our framework and enhance the expressiveness of the type system.

2.6.5 Typing expressions

The typing judgement $\Gamma; \Delta \vdash E : T$ denotes that expression E is given type T under environment $\Gamma; \Delta$. The typing rules for expressions are given in Table 2.12.

Rule (Exp Subsum) is a standard subsumption rule for expressions: if E can be given type T, then it can be conservatively given any supertype of T. In rule (Exp Split) we exploit the logic to keep track of the performed pair splitting operation and make type-checking more precise; a similar treatment applies also to (Exp Match) and (Exp Eq). Rule (Exp Assert) is standard and requires an asserted

(Exp Subsum)

$$\frac{\begin{array}{c}\Gamma;\Delta_1 \vdash E : T \qquad \Gamma;\Delta_2 \vdash T <: T' \\ \Gamma;\Delta \hookrightarrow \Gamma;\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta \vdash E : T'}$$

(Exp Appl)

$$\frac{\begin{array}{c}\Gamma;\Delta_1 \vdash M : x : T \to U \qquad \Gamma;\Delta_2 \vdash N : T \\ \Gamma;\Delta \hookrightarrow \Gamma;\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta \vdash M\ N : U\{N/x\}}$$

(Exp Let)

$$\frac{\begin{array}{c}E \rightsquigarrow^{\emptyset} [\Delta' \mid D] \qquad \Gamma;\Delta_1 \vdash D : T \\ \Gamma, x : \psi(T);\Delta_2, \mathit{forms}(x : T) \vdash E' : U \qquad x \notin \mathit{fv}(U) \\ \Gamma;\Delta,\Delta' \hookrightarrow \Gamma;\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta \vdash \mathsf{let}\ x = E\ \mathsf{in}\ E' : U}$$

(Exp Split)

$$\frac{\begin{array}{c}\Gamma;\Delta_1 \vdash M : x : T * U \\ \Gamma, x : \psi(T), y : \psi(U);\Delta_2, \mathit{forms}(x : T), \mathit{forms}(y : U), !((x,y) = M) \vdash E : V \\ \{x,y\} \cap \mathit{fv}(V) = \emptyset \qquad \Gamma;\Delta \hookrightarrow \Gamma;\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta \vdash \mathsf{let}\ (x,y) = M\ \mathsf{in}\ E : V}$$

(Exp Match)

$$\frac{\begin{array}{c}\Gamma;\Delta_1 \vdash M : T \\ \Gamma, x : \psi(H);\Delta_2, \mathit{forms}(x : H), !(h\ x = M) \vdash E : U \\ \Gamma;\Delta_2 \vdash E' : U \\ (h, H, T) \in \{(\mathsf{inl}, T_1, T_1 + T_2), (\mathsf{inr}, T_2, T_1 + T_2), (\mathsf{fold}, T'\{\mu\alpha.T'/\alpha\}, \mu\alpha.T')\} \\ \Gamma;\Delta \hookrightarrow \Gamma;\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta \vdash \mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ E\ \mathsf{else}\ E' : U}$$

(Exp Eq)

$$\frac{\begin{array}{c}\Gamma;\Delta_1 \vdash M : T \qquad \Gamma;\Delta_2 \vdash N : U \qquad x \notin \mathit{fv}(M) \cup \mathit{fv}(N)) \\ \Gamma;\Delta \hookrightarrow \Gamma;\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta \vdash M = N : \{x : \mathsf{bool} \mid !(x = \mathsf{true} \multimap M = N)\}}$$

(Exp Assume)

$$\frac{\Gamma;\Delta, F \vdash \mathsf{assume}\ \mathbf{1} : T \qquad F \neq \mathbf{1}}{\Gamma;\Delta \vdash \mathsf{assume}\ F : T}$$

(Exp True)

$$\frac{\Gamma;\Delta \vdash \diamond}{\Gamma;\Delta \vdash \mathsf{assume}\ \mathbf{1} : \mathsf{unit}}$$

(Exp Assert)

$$\frac{\Gamma;\Delta \vdash F}{\Gamma;\Delta \vdash \mathsf{assert}\ F : \mathsf{unit}}$$

(Exp Res)

$$\frac{\begin{array}{c}E \rightsquigarrow^{a} [\Delta' \mid D] \qquad \Gamma, a \updownarrow T;\Delta,\Delta' \vdash D : U \\ a \notin \mathit{fn}(U)\end{array}}{\Gamma;\Delta \vdash (\nu a)E : U}$$

(Exp Send)

$$\frac{\Gamma;\Delta \vdash M : T \qquad (a \updownarrow T) \in \Gamma}{\Gamma;\Delta \vdash a!M : \mathsf{unit}}$$

(Exp Recv)

$$\frac{\Gamma;\Delta \vdash \diamond \qquad (a \updownarrow T) \in \Gamma}{\Gamma;\Delta \vdash a? : T}$$

(Exp Fork)

$$\frac{\begin{array}{c}E_1 \rightsquigarrow^{\emptyset} [\Delta_1 \mid D_1] \qquad E_2 \rightsquigarrow^{\emptyset} [\Delta_2 \mid D_2] \\ \Gamma;\Delta'_1 \vdash D_1 : T_1 \qquad \Gamma;\Delta'_2 \vdash D_2 : T_2 \\ \Gamma;\Delta,\Delta_1,\Delta_2 \hookrightarrow \Gamma;\Delta'_1,\Delta'_2\end{array}}{\Gamma;\Delta \vdash E_1 \Rsh E_2 : T_2}$$

Table 2.12: Typing rules for expressions

$$\text{(Extr Fork)}\quad \frac{E_1 \leadsto^{\widetilde{a}} [\Delta_1 \mid D_1] \qquad E_2 \leadsto^{\widetilde{a}} [\Delta_2 \mid D_2]}{E_1 \Rsh E_2 \leadsto^{\widetilde{a}} [\Delta_1, \Delta_2 \mid D_1 \Rsh D_2]} \qquad \text{(Extr Let)}\quad \frac{E_1 \leadsto^{\widetilde{a}} [\Delta \mid D_1]}{\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \leadsto^{\widetilde{a}} [\Delta \mid \mathsf{let}\ x = D_1\ \mathsf{in}\ E_2]}$$

$$\text{(Extr Res)}\quad \frac{E \leadsto^{a,\widetilde{b}} [\Delta \mid D]}{(\nu a)E \leadsto^{\widetilde{b}} [\Delta \mid (\nu a)D]} \qquad \text{(Extr Assume)}\quad \frac{F \neq \mathbf{1} \qquad \mathit{fn}(F) \cap \{\widetilde{a}\} = \emptyset}{\mathsf{assume}\ F \leadsto^{\widetilde{a}} [F \mid \mathsf{assume}\ \mathbf{1}]} \qquad \text{(Extr Exp)}\quad \frac{\text{no other rule applies}}{E \leadsto^{\widetilde{a}} [\emptyset \mid E]}$$

Table 2.13: Extraction

formula F to be derivable from the formulas collected by the typing environment.

The most complex rule is (Exp Fork): intuitively, when type-checking the parallel expressions $E_1 \Rsh E_2$, assumptions in E_1 can be used to type-check assertions in E_2 and vice-versa. On the other hand, we need to prevent an affine assumption in E_1 from being used twice to justify assertions in both E_2 *and* E_1. This is achieved through the *extraction* relation, i.e., through the premises of the form $E_i \leadsto [\Delta_i \mid D_i]$: the extraction operation destructively collects all the assumptions from the expression E_i and returns the expression D_i obtained by purging E_i of its assumptions. The typing environment is then extended with the collected assumptions and partitioned to type-check the purged expressions D_1 and D_2 respectively. For instance, we can show that the expression $\mathsf{assume}\ F \Rsh \mathsf{assert}\ F$ is well-typed, while the expression $(\mathsf{assume}\ F \Rsh \mathsf{assert}\ F) \Rsh \mathsf{assert}\ F$ is not: in fact, notice that the latter is not safe according to Definition 2.5.1.

The extraction relation is formally defined in Table 2.13. Notice that we prevent formulas containing free names from being extracted outside of the scope of the respective binders to avoid name clashing. The extraction relation is similarly used to type-check any expression possibly containing "active" assumptions, i.e., lets, restrictions, and assumptions themselves.

2.6.6 Formal results

The main soundness results of our type system are given below.

Theorem 2.6.1 (Safety). *If $\varepsilon; \emptyset \vdash E : T$, then E is safe.*

Theorem 2.6.2 (Robust Safety). *If $\varepsilon; \emptyset \vdash E : \mathsf{Un}$, then E is robustly safe.*

Theorem 2.6.2 above and Theorem 2.4.1 (establishing the soundness of exponential serialization) constitute the two building blocks of our static verification technique, which we may finally summarize as follows.

Given any expression E, we identify the payload formulas assumed in E, and construct the corresponding exponential serializers $S_1, \ldots, S_n$ for those formulas.

Let then $E^\star = \mathsf{assume}\ S_1 \otimes \cdots \otimes S_n \upharpoonright E$. By Theorem 2.6.2, if $\varepsilon; \emptyset \vdash E^\star : \mathsf{Un}$, then $E^\star$ is robustly safe. By Theorem 2.4.1, so is the original expression E, provided that a further invariant holds for $E^\star$, namely that all multisets of formulas assumed during the evaluation of $E^\star$ are guarded.

While this latter invariant is not enforced by our type system, the desired guarantees may be achieved by requiring that the assumption of control formulas be confined within system code packaged into library functions, providing certified access and management of the capabilities associated with those formulas. The certification of the system code provided by the library function, in turn, may be achieved with limited effort, based on the syntactic guardedness condition provided by Proposition 2.4.2.

2.6.7 Encoding affine types

Here we discuss how we can take advantage of exponential serialization to encode affine types and, thus, enhance the expressiveness of our type system. For the sake of simplicity, we focus on the encoding of affine pairs.

Consider the typing environment $\Gamma; \Delta \triangleq x : \mathsf{Un}, y : \mathsf{Un}; A(x), B(y)$. Standard refinement type systems as [10] allow for the following type judgement:

$$\Gamma; \Delta \vdash (x, y) : \{x : \mathsf{Un} \mid A(x)\} * \{y : \mathsf{Un} \mid B(y)\}$$

If the formulas $A(x)$ and $B(y)$ are interpreted as affine resources, however, the previous type assignment is sound only as long as the pair (x, y) can be split only once, since every application of rule (Exp Split) for pair destruction introduces the formulas $A(x), B(y)$ into the typing environment. Since our type system does not feature affine types and has no way to enforce a single deconstruction of a pair, it conservatively forbids the previous type judgement, in that the premises of rule (Val Pair) require an exponential typing environment.

Nevertheless, the following type judgement is allowed by our type system:

$$x : \mathsf{Un}, y : \mathsf{Un}; A(x), B(y), S_1, S_2 \vdash (x, y) : \{x : \mathsf{Un} \mid A'(x)\} * \{y : \mathsf{Un} \mid B'(y)\}$$

where $A'(x) \triangleq\ !(P_1(x) \multimap A(x))$ and $B'(y) \triangleq\ !(P_2(y) \multimap B(y))$ are the serialized variants of $A(x)$ and $B(y)$ respectively, while $S_1 \triangleq\ !\forall x.(A(x) \multimap A'(x))$ and $S_2 \triangleq$ $!\forall y.(B(y) \multimap B'(y))$ are the corresponding serializers. Here, the main idea for type-checking is to appeal to environment rewriting to consume the affine formulas $A(x)$ and $B(y)$, and introduce their exponential counterparts $A'(x)$ and $B'(y)$ into the typing environment before assigning a type to the components of the pair.

The interesting point now is that the pair (x, y) can be split arbitrarily often, but the affine formulas $A(x)$ and $B(y)$ can be retrieved at most once, as long as the control formulas $P_1(x)$ and $P_2(y)$ are assumed at most once in the application code. In this way, we recover the expressiveness provided by affine types. We actually even go beyond that, allowing for a liberal usage of the value itself, as opposed to enforcing the affine usage of any data structure which contains an affine component, as dictated by many earlier substructural frameworks.

2.6.8 Encoding cryptography

Formal cryptography can be encoded inside RCF in terms of *sealing* [61, 76]. A *seal* for a type T is a pair of functions: a sealing function $T \to \mathsf{Un}$ and an unsealing function $\mathsf{Un} \to T$. Intuitively, for symmetric cryptography, these functions model encryption and decryption operations, respectively. A payload of type T can be sealed to type Un and sent over the untrusted network; conversely, a message retrieved from the network with type Un can be unsealed to its correct type T. This mechanism is implemented in terms of a list of pairs, which is stored in a global reference that can only be accessed using the sealing and unsealing functions. Upon sealing, the payload p is paired with a fresh, public value h (the *handle*) representing its sealed version, and the pair (p, h) is stored in the list; conversely, the unsealing function looks for the handle h in the list and returns the associated payload p.

Since for symmetric cryptography the possession of the key allows to perform both encryption and decryption operations, for this cryptographic scheme we identify the key with the seal, i.e., we give access to both the sealing and the unsealing function to any owner of the key. Different cryptographic primitives, like public key encryptions and signature schemes, can be encoded following the same recipe: for instance, a signing key may consist of both the sealing and the unsealing functions, and be given type $\mathsf{SigKey}(T) \triangleq (T \to \mathsf{Un}) * (\mathsf{Un} \to T)$. The corresponding verification key, instead, comprises only the unsealing function and is given type $\mathsf{VerKey}(T) \triangleq \mathsf{Un} \to T$. The functions sign and verify introduced in Section 2.2.3 can then be straightforwardly implemented: $\mathsf{sign}\ M\ x_k$ just extracts the first component of x_k and calls it with parameter M, while $\mathsf{verify}\ N\ x_{vk}$ simply invokes x_{vk} with parameter N.

One crucial benefit of our exponential serialization technique is that we can immediately leverage the sealing-based cryptographic library proposed by Bengtson et al. [10]. The reason is that we never apply cryptographic operations directly on messages with affine refinements, but we rely on exponentially serialized versions of such refinements. Without the serialization approach, we would need to define a different implementation of the sealing/unsealing functions: namely, we would have to enforce that an affine payload is never extracted more than once from the list stored in the global reference, and the unsealing function would have to remove the payload from the secret list. This would complicate the sealing-based abstraction of cryptography and require additional reasoning to justify its soundness. Instead, with our approach, the unsealing function does not need to be changed: we can invoke it an arbitrarily number of times to retrieve the payload, but the associated refinements will be retrieved at most once through exponential serialization.

2.6.9 Algorithmic type-checking

Our type system includes several sources of non-determinism, which make it hard to implement an efficient type-checker. Still, we can devise an algorithmic variant of our type system, and prove it sound and complete. This contribution is due to Fabienne Eigner and Matteo Maffei, hence we only summarize the intuition for the sake of completeness. Additional details, including proofs, can be found in [21].

In the algorithmic system standard sources of non-determinism, like subtyping or refined value types, are eliminated using type annotations. The rewriting of logical environments $\Gamma; \Delta \hookrightarrow \Gamma; \Delta'$, which is the distinctive source of non-determinism of our system, is harder to deal with. The core idea is to dispense altogether with logical environments and to construct bottom-up a single logical formula that characterizes all the proof obligations that would normally be introduced along the typing derivation. In such a way, all the burden due to resource management can be shifted to an external theorem prover such as `llprover` [80], which would need to deal with this issue anyway. Typing an expression algorithmically then constitutes of two steps:

1. The expression (decorated with type annotations whenever needed) is type-checked using the algorithmic type system. This process is syntax-directed and in case of success yields *one* proof obligation.

2. The proof obligation is verified.

If both steps succeed, then the expression is well-typed (see the soundness result below). More in detail, every typing judgement of the form $\Gamma; \Delta \vdash \mathcal{J}$ is matched by an algorithmic counterpart of the form $\Gamma \vdash_{\mathsf{alg}} \mathcal{J}; F$. For instance, the algorithmic typing rule for pairs looks as follows:

$$\textsc{(Val Pair Alg)}\quad \frac{\Gamma \vdash_{\mathsf{alg}} M : T; F_1 \qquad \Gamma \vdash_{\mathsf{alg}} N : U\{M/x\}; F_2}{\Gamma \vdash_{\mathsf{alg}} (M, N) : x : T * U; !F_1 \otimes !F_2}$$

The proof obligation associated to a pair is the conjunction of the proof obligations of its component, under the additional restriction that such formulas must be considered as exponential: this mirrors the side-condition on (Val Pair) requiring an exponential environment to type-check the premises of the rule.

The formal correspondence between the two type systems can be summarized as follows:

1. Soundness: if $\Gamma \vdash_{\mathsf{alg}} \mathcal{J}; F$ and $\Delta \vdash F$, then $\Gamma; \Delta \vdash \mathcal{J}$.

2. Completeness: if $\Gamma; \Delta \vdash \mathcal{J}$, then there exists F such that $\Gamma \vdash_{\mathsf{alg}} \mathcal{J}; F$ and $\Delta \vdash F$.

2.7 Example: electronic purchase

We consider a variant of *EPMO*, a nonce-based e-payment protocol proposed by Guttman et al. [52]. The protocol narration is informally represented in Table 2.14.

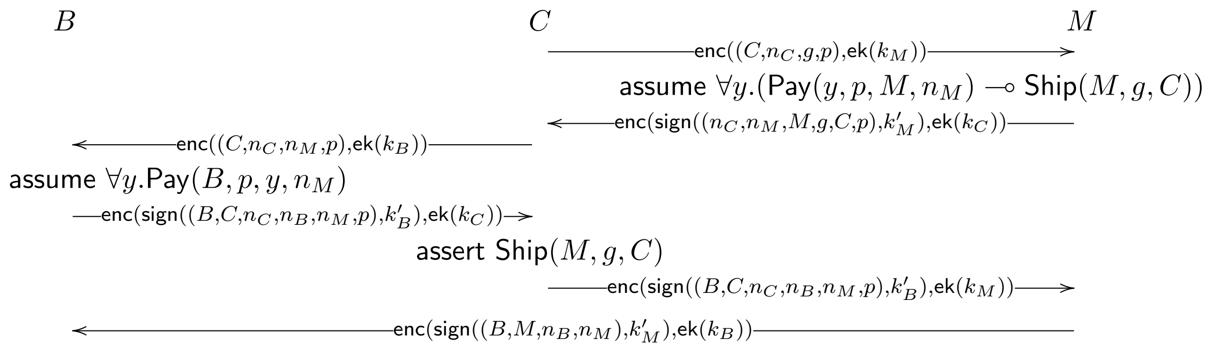
B
C
M
$\mathsf{enc}((C,n_C,g,p),\mathsf{ek}(k_M))$
$\mathsf{assume}\ \forall y.(\mathsf{Pay}(y,p,M,n_M) \multimap \mathsf{Ship}(M,g,C))$
$\mathsf{enc}(\mathsf{sign}((n_C,n_M,M,g,C,p),k'_M),\mathsf{ek}(k_C))$
$\mathsf{enc}((C,n_C,n_M,p),\mathsf{ek}(k_B))$
$\mathsf{assume}\ \forall y.\mathsf{Pay}(B,p,y,n_M)$
$\mathsf{enc}(\mathsf{sign}((B,C,n_C,n_B,n_M,p),k'_B),\mathsf{ek}(k_C))$
$\mathsf{assert}\ \mathsf{Ship}(M,g,C)$
$\mathsf{enc}(\mathsf{sign}((B,C,n_C,n_B,n_M,p),k'_B),\mathsf{ek}(k_M))$
$\mathsf{enc}(\mathsf{sign}((B,M,n_B,n_M),k'_M),\mathsf{ek}(k_B))$

Table 2.14: A variant of the *EPMO* protocol

Initially, a customer C contacts a merchant M to buy some goods g for a given price p; the request is encrypted under the public key of the merchant, $\mathsf{ek}(k_M)$, and includes a fresh nonce, n_C. If M agrees to proceed in the transaction by providing a signed response, C informs her bank B to authorize the payment. The bank replies by providing C a receipt of authorization, called the *money order*, which is then forwarded to M. Now M can verify that C is entitled to pay for the goods and complete the transaction by sending a signed request to B to cash the money order. At the end of the run, the bank transfers the funds and the merchant ships the goods to the customer.

A peculiarity of the protocol is that the identifier n_C is employed by C to authenticate *two* different messages, namely the replies by M and B. This pattern cannot be validated by most existing type systems, since the mechanisms hard-coded therein to deal with nonce-handshakes enforce the freshness of each nonce to be checked only once. Our framework, instead, allows for a very natural treatment of such authentication pattern, whose implementation can be written mostly oblivious of the security verification process based on lightweight logical annotations. For the sake of simplicity, we focus only on the aspects of the verification connected to the guarantees provided to C, which are the most interesting to prove.

We define two predicates used in the analysis: $\mathsf{Pay}(B,p,M,n_M)$ states that B authorizes the payment p to M in reference to the order identified by n_M, while $\mathsf{Ship}(M,g,C)$ formalizes that M can ship the goods g to C. The protocol code for the customer, enriched with the most relevant type annotations and the serializers, is shown below. For the sake of readability, we use F#-like syntax and some syntactic sugar like tuples and pattern matching to present code snippets from our example: these can be encoded in RCF using standard techniques [10].

```
// Serializer for M (needed to type-check M)
assume !forall xp,xM,xnM,xg,xC,xnC.
       (forall y.(Pay(y,xp,xM,xnM) --o Ship(xM,xg,xC)) --o
       !(N1(xnC) --o (forall y.(Pay(y,xp,xM,xnM) --o Ship(xM,xg,xC))))

// Serializer for B (needed to type-check B)
assume !forall yB,yp,ynC,ynM.
       (forall y.(Pay(yB,yp,y,ynM)) --o
       !(N2(ynC) --o (forall y.(Pay(yB,yp,y,ynM))))

// Type of the message from M to C
type MsgMC = MsgMC of (xnC: Un * xnM: Un * xM: Un * xg: Un * xC: Un * xp: Un)
  {!(N1(xnC) --o forall y.(Pay(y,xp,xM,xnM) --o Ship(xM,xg,xC))}

// Type of the message from B to C
type MsgBC = MsgBC of (yB: Un * yC: Un * ynC: Un * ynB: Un * ynM: Un * yp: Un)
  {!(N2(ynC) --o forall y.(Pay(yB,yp,y,ynM))}

let (mkTid : unit -> {x: bytes | N1(x) times N2(x)}) () =
  let xf = mkFresh () in assume (N1(xf) times N2(xf)); xf

let cust C addC M addM B addB g p kC ekM ekB
         (vkM: (MsgMC, MsgMB) either VerKey) (vkB: MsgBC VerKey) =
  let nC = mkTid () in
  let msgCM1 = encrypt (C, nC, g, p) ekM in send addM msgCM1;
  let signMC = decrypt (receive addC) kC in
  let plainMC = verify signMC vkM in
  match plainMC with MsgMC (=nC, xnM, =M, =g, =C, =p) ->
      let msgCB = encrypt (C, nC, xnM, p) ekB in send addB msgCB;
      let signBC = decrypt (receive addC) kC in
      let plainBC = verify signBC vkB in
        match plainBC with MsgBC (=B, =C, =nC, xnB, =xnM, =p) ->
          assert Ship(M, g, C);
          let msgCM2 = encrypt signBC ekM in send addM msgCM2
```

Initially, we let the customer call the library function mkTid, which generates a fresh transaction identifier, corresponding to n_C in the protocol specification, and provides via its return type two distinct capabilities $\mathsf{N}_1(n_C)$ and $\mathsf{N}_2(n_C)$, later employed to authenticate two different messages received by C. Since the signing key of M is used to certify messages of two different types, at steps 2 and 6 of the protocol, the corresponding verification key available to the customer through the variable vkM refers to a sum type. We present only the MsgMC component of such type, since it is the one needed to type-check the code of C: the corresponding refined formula in the type definition describes the promise by M to ship the goods as soon as the requested payment has been authorized by any bank. We then use vkB to convey the other formula which is needed to type-check C, namely a statement that B authorizes the payment to any merchant to whom C wishes to transfer the money order. The hypotheses collected by C are enough to prove her assertion, i.e., to be sure that the request by M has been fulfilled and the goods will be shipped, hence the implementation is well-typed.

The proof obligation assigned to the `cust` function by the algorithmic formulation of our type system is shown below:

```
∀C.∀M.∀B.∀G.∀p.
  ∀nC.((N1(nC) ⊗ N2(nC)) ⊸
    ∀xnM.(!(N1(nC) ⊸ (∀y.Pay(y,p,M,xnM) ⊸ Ship(M,g,C))) ⊸
         !(N2(nC) ⊸ (∀z.Pay(B,p,z,xnM))) ⊸
            Ship(M,g,C)))
```

For the sake of readability we removed all unnecessary occurrences of **1** and unused quantified variables. In this example, as well as in the other protocol we considered, the problem of solving equalities is reduced to the unification of variables. This allows us to use the `llprover` [80] theorem prover, which at the moment does not support equality theories. The above formula is discharged in less than 20 ms.

2.8 Example: Kerberos

In the *EPMO* protocol presented before, the nonce n_C is checked twice by the customer C and plays the role of a transaction identifier. Interestingly, there are protocols where such identifiers are not just checked multiple times, but also by different parties. This is exactly the case for the Kerberos protocol [74] shown in Table 2.15 below.

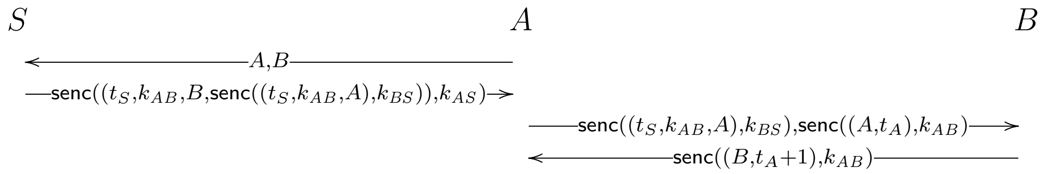
S
A
B
A,B
$\mathsf{senc}((t_S,k_{AB},B,\mathsf{senc}((t_S,k_{AB},A),k_{BS})),k_{AS})$
$\mathsf{senc}((t_S,k_{AB},A),k_{BS}),\mathsf{senc}((A,t_A),k_{AB})$
$\mathsf{senc}((B,t_A+1),k_{AB})$

Table 2.15: Kerberos protocol

The protocol establishes a fresh session key k_{AB} between principals A and B through a trusted server S. Kerberos employs timestamps like t_S and t_A to prove session recentness and protect against replay attacks. In this case, the timestamp t_S is checked by both A and B to ensure that k_{AB} is fresh.

In our implementation, we build on a very simple library for timestamp management, that we allow the principals to access. Of course, we could consider more realistic implementations, but the following one suffices to convey the intuition.

```
let get_ts r () =
    r := !r + 1; !r

let check_ts r t id =
    if (t > !r) then r := t; assume F(id,t)
    else failwith "not a fresh timestamp"

let init_ts =
    let r = ref 0 in
    (get_ts r, check_ts r)
```

Each principal stores the last received timestamp in the reference `r`, created through an invocation to the function `init_ts`. The goal of the function is to protect such a reference to guarantee a correct timestamp management: indeed, the reference is accessible only through the functions `get_ts` and `check_ts` returned by `init_ts`. The function `get_ts`: $\mathsf{Ref\ int} \to \mathsf{unit} \to \mathsf{int}$ is used to create fresh timestamps, while the dependent function `check_ts`: $\mathsf{Ref\ int} \to x : \mathsf{int} \to id : \mathsf{int} \to \{\mathsf{F}(id, x)\}$ is used to check whether a received timestamp x is fresh and can be used to communicate with participant id or not. The code of the function performs a conditional branch: if the timestamp is new, it *assumes* the logical formula encoding such a fact; otherwise, it fails. The function `failwith` throws an exception, so it can be safely given the polymorphic type $\mathsf{int} \to \alpha$; as a consequence, `check_ts` can be given the previous dependent function type, whose return type provides the necessary freshness assumption.

We let the server S assume the predicates $\mathsf{K_A}(B, k_{AB})$ and $\mathsf{K_B}(A, k_{AB})$ upon creation of the session key k_{AB}, to model that k_{AB} is a fresh session key which can be used by A to communicate with B and vice-versa. The code of the principal A looks as follows:

```
// Definition of Payload omitted
type MsgSA = MsgSA of (xtS: int * xkAB: Payload Symkey * xB: Un * y: Un)
  {!(F(xB,xtS) --o kA(xB,xkAB))}

// We omit refinements in the next two types for the sake of a simple example
type MsgAB = MsgAB of (A: un * tA: int)
type MsgBA = MsgBA of (B: un * tB: int)

let initiator A addA B addB S addS (kAS: MsgSA SymKey) =
    let (get_ts, check_ts) = init_ts () in
    send addS (A,B);
    let msgSA = receive addA in
    let plainSA = sym_decrypt msgSA kAS in
    match plainSA with
      MsgSA (xtS, xkAB, xB, y) ->
        let xtS = check_ts xtS xB in
```

```
assert kA(xB,xkAB);
let tA = get_ts in
let msgAB = sym_encrypt (MsgAB (A, tA)) xkAB in
send addB (y,msgAB);
let msgBA = receive addA in
let plainBA = sym_decrypt msgBA xkAB in
match plainBA with
  MsgBA (=B, =tA+1) ->
    let _ = check_ts tA+1 B then
      ... // use the key xkAB
```

Let $T_{AB} = \mathsf{SymKey}(\mathsf{Payload})$ be the type of the session key k_{AB}, then the type of k_{AS} is $\mathsf{SymKey}(x_t : \mathsf{Un} * x_k : T_{AB} * x_B : \mathsf{Un} * \{y : \mathsf{Un} \mid !(\mathsf{F}(x_B, x_t) \multimap \mathsf{K_A}(x_B, x_k))\})$. We use this key to convey the formula $\mathsf{K_A}(B, k_{AB})$ from S to A. Since this formula is affine, it is encapsulated using our serialization technique with the control formula $\mathsf{F}(x_B, x_t)$ that denotes that a timestamp x_t is fresh and can be used in a communication with x_B. We can give k_{BS} a similar type, i.e., $\mathsf{SymKey}(x_t : \mathsf{Un} * x_k : T_{AB} * x_A : \mathsf{Un} * \{y : \mathsf{Un} \mid !(\mathsf{F}(x_A, x_t) \multimap \mathsf{K_B}(x_A, x_k))\})$. Note that, by enriching `MsgAB` and `MsgBA` with suitable refinements, we can statically verify also the authentication guarantees established between A and B and perform a complete analysis of the protocol.

2.9 Related work

Several papers develop type systems for (variants of) RCF [12, 10, 41, 8, 78] but, with the exception of F* [78], they do not support resource-aware policies: in fact, even for simple linearity properties like injective agreement they rely on hand-written proofs [11].

F* [78] is a dependently typed functional language for secure distributed programming, featuring refinement types to reason about authorization policies and affine types to reason about stateful computations on affine *values*. Similarly to companion proposals for RCF, the type system of F* assumes the existence of the contraction rule in the underlying logic, hence, it does not support authorization policies built over affine *formulas*. While some simple authentication patterns (e.g., basic nonce handshakes) may certainly be expressed by encoding affine predicates in terms of affine values, other more complex authentication mechanisms are much harder to handle in these terms. The *EPMO* protocol we analyze in Section 2.7 provides one such case, as (*i*) the nonce it employs may not be construed as an affine value because it is used twice, and (*ii*) the logical formulae justified by cryptographic message exchanges are more structured than simple predicates. Though it might be possible to come up with sophisticated encodings of these authentication mechanisms in the programming language (by resorting to, e.g., pairs of affine tokens to encode a double usage of the same nonce and special functions to eliminate logical implications), such encodings are hard to formulate in a general manner and, we argue, are much better expressed in terms of policy annotations than in some ad-hoc programming pattern.

Bhargavan et al. [13] propose a technique for the verification of F# protocol implementations by automatically extracting ProVerif models [16], using an extension of the functions-as-processes encoding proposed by Milner [60]. Remarkably, the analysis can deal with injective agreement. On the other hand, the analysis carried out with ProVerif is not modular and has been shown less robust and scalable than type-checking [12]. Furthermore, the fragment of F# considered is rather restrictive: for instance, it does not include higher-order functions and admits only very limited uses of recursion and state.

A formal account on the integration of refinement types and substructural logics was first proposed by Mandelbaum et al. [59] with a system for local reasoning about program state built around a fragment of intuitionistic linear logic. Later, Bierhoff and Aldrich developed a framework for modular type-state checking of object-oriented programs [15, 77, 63]. None of these systems deals with the presence of hostile (or untyped) program components, or attackers, a feature that is instead distinctive of our system: adapting the previous frameworks to take into account interactions with an untyped context would require fundamental changes to their typing rules. The original RCF type-checker [10], for instance, employs a security-oriented kinding relation to reason about messages sent to and received from the attacker, which we also adopt in our type system. Recent variants of the RCF type-checker dispense with the kinding relation and even with concurrency [78], but they rely on manually proven logical invariants capturing security properties of the cryptographic library and, in some cases, of the protocol itself.

Tov and Pucella [81] have recently shown how to use behavioral contracts to link code written in an affine language to code in a conventionally typed language. The idea is to coerce affine values to non-affine ones that can be shared with the context, but can still be reasoned about safely using dynamic access counts. There are intriguing similarities between this approach and the usage of nonces and session keys to enforce linearity properties in an adversarial setting, which are worth to be investigated in the future. The two type systems are, however, fundamentally different, since our present work deals with an affine refinement logic and an adversarial setting, which makes a precise comparison hard to formulate.

There exist a number of types and effects systems targeted at the analysis of authenticity properties of cryptographic protocols [50, 51, 23]. These type systems incorporate ad-hoc mechanisms to deal with nonce handshakes and, thus, to enforce injective agreement properties. The exponential serialization technique can be seen as a logic-based generalization of such mechanisms, independent of the language and type system. As a consequence, our type system is similarly able to verify authenticity in terms of injective agreement, while allowing for expressing also a number of more sophisticated properties involving access counts and usage bounds. As a downside, the current formulation of our type system does not allow to validate some specific nonce-handshake idioms, like the SOSH scheme [51]. Still, this can be recovered by extending our type system with union and intersection types, as shown in [8].

In a previous work [19, 21], we made initial steps towards the design of a sound system for resource-sensitive authorization, drawing on techniques from typing systems for authentication and an affine extension of existing refinement typing systems for the applied pi-calculus [5]. That work aims at analyzing cryptographic protocols as opposed to their implementations. Furthermore, the type system is there designed around a specific cryptographic library: the consequence is that extending the analysis to new primitives requires significant changes in the soundness proof of the type system. In contrast, the usage of a λ-calculus in this work allows us to encode cryptography in the language using a standard sealing mechanism (cf. Section 2.6.8), which makes the analysis technique easily extensible to new cryptographic primitives. Finally, the non-standard nature of our previous type system makes it difficult to devise an efficient algorithmic variant.

Chapter 3

Proofs of Chapter 2

3.1 Soundness of exponential serialization

3.1.1 Preliminaries

We first introduce some notational conventions. We let:

$$\hat{S} \in \{!\forall \widetilde{x}.(P \multimap !(C \multimap P)), \forall \widetilde{x}.(P \multimap !(C \multimap P))\}$$

for some (possibly empty) $\widetilde{x}$ and some P, C. We also write Δ, F^n as a short for the multiset $\Delta, F, \ldots, F$ (n times).

Definition 3.1.1 (Well-formation)**.** A multiset of formulas $\Delta = \Delta_1, \Delta_2, \Delta_3$ is *well-formed* if and only if it is stratified and $\Delta_1 = P_1, \ldots, P_l$, $\Delta_2 = G_1, \ldots, G_m$, $\Delta_3 = \hat{S}_1, \ldots, \hat{S}_n$.

We define a partial function *guard* from formulas to control formulas, defined in the following cases:

- $\mathit{guard}(C \multimap P) = C$;
- $\mathit{guard}(P \multimap G) = \mathit{guard}(G)$;
- $\mathit{guard}(\forall x.F) = \mathit{guard}(F)$ whenever $\mathit{guard}(F)$ is defined;
- $\mathit{guard}(!F) = \mathit{guard}(F)$ whenever $\mathit{guard}(F)$ is defined.

We extend the notion of *rank* to a multiset of formulas Δ as follows:

$$\mathit{rk}(\Delta) = \mathit{min}\ \{\mathit{rk}(C) \mid \exists F \in \Delta : \mathit{guard}(F) = C\}$$

If the previous set is empty, we stipulate $\mathit{rk}(\Delta) = +\infty$.

A control formula C is *active* in Δ if and only if $\mathit{rk}(C) \leq \mathit{rk}(\Delta)$. We simply say that C is active whenever Δ is clear from the context.

Definition 3.1.2 (Weak Guardedness)**.** A well-formed multiset Δ is *weakly guarded* if and only if, for every active control formula C, we have that $\Delta \vdash C^k$ implies $k \leq 1$.

We note as expected that any guarded multiset is also weakly guarded.

Proposition 3.1.1. *If Δ is guarded, then it is weakly guarded.*

In the next results we focus without loss of generality on cut-free proofs.

3.1.2 Main results

Lemma 3.1.2. *Let $\Delta = \Delta', P \multimap !(C \multimap P)$ be weakly guarded and let C be active in Δ. If $\Delta \vdash P'$, then $\Delta' \vdash P'$.*

Proof. By induction on the derivation of $\Delta \vdash P'$:

Case (Ident): this rule cannot be applied, since $P \multimap !(C \multimap P)$ is not a payload formula;

Case (Weak): if the principal formula is $P \multimap !(C \multimap P)$, the conclusion is immediate. Otherwise, let $\Delta' = \Delta'', F$ and let $\Delta \vdash P'$ by the hypothesis $\Delta'', P \multimap !(C \multimap P) \vdash P'$. By Lemma 3.1.9 we know that the latter multiset is weakly guarded and C is active by Lemma 3.1.8, so we can apply the inductive hypothesis to get $\Delta'' \vdash P'$ and conclude $\Delta' \vdash P'$ by (Weak);

Case (Contr): let $\Delta' = \Delta'', !F$ and let $\Delta \vdash P'$ by the hypothesis $\Delta, !F \vdash P'$. By Lemma 3.1.9 we know that the latter multiset is weakly guarded and C is active by Lemma 3.1.8, so we can apply the inductive hypothesis to get $\Delta', !F \vdash P'$ and conclude $\Delta' \vdash P'$ by (Contr);

Case ($\otimes$-Left): let $\Delta' = \Delta'', F_1 \otimes F_2$ and let $\Delta \vdash P'$ from the hypothesis $\Delta'', F_1, F_2, P \multimap !(C \multimap P) \vdash P'$. By Lemma 3.1.9 we know that the latter multiset is weakly guarded and C is active by Lemma 3.1.8, so we can apply the inductive hypothesis to get $\Delta'', F_1, F_2 \vdash P'$ and conclude $\Delta' \vdash P'$ by ($\otimes$-Left);

Case ($\otimes$-Right): let $\Delta \vdash P_1 \otimes P_2$ by the hypotheses $\Delta_1 \vdash P_1$ and $\Delta_2 \vdash P_2$ with $\Delta = \Delta_1, \Delta_2$. By Lemma 3.1.9 both Δ_1 and Δ_2 are weakly guarded and C is active in both by Lemma 3.1.8, so we can apply the inductive hypothesis to either $\Delta_1 \vdash P_1$ or $\Delta_2 \vdash P_2$, according to whether $P \multimap !(C \multimap P)$ occurs in Δ_1 or in Δ_2, and conclude by ($\otimes$-Right);

Case ($\multimap$-Left): we distinguish two cases, according to the principal formula:

- let $\Delta' = \Delta'', F_1 \multimap F_2$ and let $\Delta \vdash P'$ by the hypotheses $\Delta_1 \vdash F_1$ and $\Delta_2, F_2 \vdash P'$ with $\Delta'', P \multimap !(C \multimap P) = \Delta_1, \Delta_2$. By Lemma 3.1.9 we have that both Δ_1 and Δ_2, F_2 are weakly guarded and C is active in both by

Lemma 3.1.8. Moreover, we note that F_1 is a payload formula. We are thus allowed to apply the inductive hypothesis to either $\Delta_1 \vdash F_1$ or $\Delta_2, F_2 \vdash P'$, according to whether $P \multimap !(C \multimap P)$ occurs in Δ_1 or in Δ_2, and conclude by ($\multimap$-Left);

- let $\Delta \vdash P'$ by the hypotheses $\Delta_1 \vdash P$ and $\Delta_2, !(C \multimap P) \vdash P'$ with $\Delta' = \Delta_1, \Delta_2$. By Lemma 3.1.9 we have that $\Delta_2, !(C \multimap P)$ is weakly guarded and C is active there by Lemma 3.1.8, so by Corollary 3.1.17 we have $\Delta_2, P \vdash P'$. We can then conclude $\Delta_1, \Delta_2 \vdash P'$ by a standard Cut Elimination argument;

Case ($\multimap$-Right): the only possibility is that $\Delta \vdash B_1 \multimap B_2$ by the hypothesis $\Delta, B_1 \vdash B_2$. By Lemma 3.1.12 we know that Δ, B_1 is weakly guarded and C is active, so by inductive hypothesis we get $\Delta', B_1 \vdash B_2$ and we conclude $\Delta' \vdash B_1 \multimap B_2$ by ($\multimap$-Right);

Case ($\forall$-Left): let $\Delta' = \Delta'', \forall x.F$ and let $\Delta \vdash P'$ from $\Delta'', F\{t/x\}, P \multimap !(C \multimap P) \vdash P'$ for some t. By Lemma 3.1.9 we know that the latter multiset is weakly guarded and C is active by Lemma 3.1.8, so we can apply the inductive hypothesis to get $\Delta'', F\{t/x\} \vdash P'$ and conclude $\Delta' \vdash P'$ by ($\forall$-Left);

Case ($\forall$-Right): the only possibility is that $\Delta \vdash \forall x.B$ by $\Delta \vdash B$ with $x \notin fv(\Delta)$. By inductive hypothesis $\Delta' \vdash B$, so we conclude $\Delta' \vdash \forall x.B$ by ($\forall$-Right);

Case (!-Left): let $\Delta' = \Delta'', !F$ and let $\Delta \vdash P'$ by the hypothesis $\Delta'', F, P \multimap !(C \multimap P) \vdash P'$. By Lemma 3.1.9 we know that the latter multiset is weakly guarded and C is active by Lemma 3.1.8, so we can apply the inductive hypothesis to get $\Delta'', F \vdash P'$ and conclude $\Delta' \vdash P'$ by (!-Left);

Case (!-Right): this rule cannot be applied, since $P \multimap !(C \multimap P)$ is not exponential.

□

Lemma 3.1.3. *Let $\Delta = \Delta', \forall\widetilde{x}.(P \multimap !(C \multimap P))$ be weakly guarded and let C be active in Δ. If $\Delta \vdash P'$, then $\Delta' \vdash P'$.*

Proof. By induction on the derivation of $\Delta \vdash P'$, much as in the proof of Lemma 3.1.2. We show just the cases which are actually different:

Case ($\multimap$-Left): if $\widetilde{x}$ is empty, we immediately conclude by Lemma 3.1.2. Otherwise, let $\Delta' = \Delta'', F_1 \multimap F_2$ and let $\Delta \vdash P'$ by the hypotheses $\Delta_1 \vdash F_1$ and $\Delta_2, F_2 \vdash P'$ with $\Delta'', \forall\widetilde{x}.(P \multimap !(C \multimap P)) = \Delta_1, \Delta_2$. By Lemma 3.1.9 we have that both Δ_1 and Δ_2, F_2 are weakly guarded and C is active in both by Lemma 3.1.8. Moreover, we note that F_1 is a payload formula. We are thus allowed to apply the inductive hypothesis to either $\Delta_1 \vdash F_1$ or $\Delta_2, F_2 \vdash P'$, according to whether $\forall\widetilde{x}.(P \multimap !(C \multimap P))$ occurs in Δ_1 or in Δ_2, and conclude by ($\multimap$-Left);

Case ($\forall$-LEFT): we have two cases. If the principal formula is $\forall\widetilde{x}.(P \multimap !(C \multimap P))$, we have again two possibilities: either every quantified variable is instantiated by the application of the rule or not. In the first case, we conclude by Lemma 3.1.2; in the second case, the conclusion is immediate by inductive hypothesis. Otherwise, if the instantiated formula belongs to Δ', the conclusion follows by applying the inductive hypothesis and ($\forall$-LEFT).

□

Lemma 3.1.4. *Let $\Delta = \Delta', !\forall\widetilde{x}.(P \multimap !(C \multimap P))$ be weakly guarded and let C be active in Δ. If $\Delta \vdash P'$, then $\Delta' \vdash P'$.*

Proof. We prove a stronger statement, namely:

$$\forall n \geq 0 : \Delta', (!\forall\widetilde{x}.(P \multimap !(C \multimap P)))^n \vdash P' \text{ implies } \Delta' \vdash P',$$

provided that the initial hypotheses are satisfied. We proceed by induction on the derivation of the judgement in the premise, much as in the proofs of Lemmas 3.1.2 and 3.1.3, and we show just the cases which are actually different:

Case (IDENT): if $n \geq 1$, the rule cannot be applied, so the conclusion is trivial;

Case (WEAK): if the principal formula is $!\forall\widetilde{x}.(P \multimap !(C \multimap P))$, the conclusion is immediate by inductive hypothesis. Otherwise, if the principal formula belongs to Δ', the conclusion follows by applying the inductive hypothesis and (WEAK);

Case (CONTR): we have two cases. If the principal formula is $!\forall\widetilde{x}.(P \multimap !(C \multimap P))$, the conclusion is immediate by inductive hypothesis. Otherwise, if the principal formula belongs to Δ', the conclusion follows by applying the inductive hypothesis and (CONTR);

Case ($\otimes$-RIGHT): let $\Delta \vdash P_1 \otimes P_2$ by the hypotheses $\Delta_1 \vdash P_1$ and $\Delta_2 \vdash P_2$ with $\Delta = \Delta_1, \Delta_2$. By Lemma 3.1.9 both Δ_1 and Δ_2 are weakly guarded and C is active in both by Lemma 3.1.8, so we can apply the inductive hypothesis to both $\Delta_1 \vdash P_1$ and $\Delta_2 \vdash P_2$, and conclude by ($\otimes$-RIGHT);

Case (!-LEFT): we have two cases. If the principal formula is $!\forall\widetilde{x}.(P \multimap !(C \multimap P))$, we first apply the inductive hypothesis and then we conclude by Lemma 3.1.3. Otherwise, if the principal formula belongs to Δ', the conclusion follows by first applying the inductive hypothesis and then using (!-LEFT);

Case (!-RIGHT): let $\Delta \vdash !B$ by the hypothesis $\Delta \vdash B$ with Δ exponential. By inductive hypothesis $\Delta' \vdash B$, so we conclude $\Delta' \vdash !B$ by (!-RIGHT).

□

Lemma 3.1.5. *Let $\Delta' = P_1, \ldots, P_m$. If $\Delta = \Delta', S_1, \ldots, S_n$ is weakly guarded and $\Delta \vdash P'$, then $\Delta' \vdash P'$.*

Proof. By an appropriate number of applications of Lemma 3.1.4, using Proposition 3.1.6 to identify a candidate serializer to remove at each step. □

Restatement of Theorem 2.4.1. Let $\Delta' = P_1, \dots, P_m$. If $\Delta = \Delta', S_1, \dots, S_n$ is guarded and $\Delta \vdash P'$, then $\Delta' \vdash P'$.

Proof. Immediate by Lemma 3.1.5, since any guarded multiset is also weakly guarded by Proposition 3.1.1. □

Restatement of Proposition 2.4.2. If $\Delta = B_1, \dots, B_l, C_1, \dots, C_m, S_1, \dots, S_n$ is stratified and the control formulas in Δ are pairwise distinct, then Δ is guarded.

Proof. We first show that Δ is weakly guarded. Let $\Delta_1 = B_1, \dots, B_l$, $\Delta_2 = C_1, \dots, C_m$ and $\Delta_3 = S_1, \dots, S_n$. Let us assume by contradiction that $\Delta \vdash C^h$ with $h \geq 2$ for some active control formula C. By Lemma 3.1.10 we have $\Delta_2 \vdash C^h$, since any formula in Δ_1 and Δ_3 has an infinite rank, while the rank of C is finite. By Proposition 3.1.7, C must occur at least h times in Δ_2, but this is contradictory with respect to the initial hypotheses.

Now we know that Δ is weakly guarded and we can show that it is, in fact, guarded. Let us assume by contradiction that $\Delta \vdash C^h$ with $h \geq 2$ for some arbitrary control formula C, then by Lemma 3.1.5 we have $\Delta_1, \Delta_2 \vdash C^h$. By Proposition 3.1.7, C must occur at least h times in Δ_1, Δ_2, but this is contradictory with respect to the initial hypotheses. □

3.1.3 Auxiliary results

Proposition 3.1.6. *Let $\Delta = P_1, \dots, P_m, S_1, \dots, S_n$. If $n > 0$, then there exists $S_i \in \Delta$ such that $guard(S_i)$ is active in Δ.*

Proof. This immediately follows by the definitions of $rk(\Delta)$ and active control formula. □

Proposition 3.1.7. *Let $\Delta = B_1, \dots, B_l, C_1, \dots, C_m$. If $\Delta \vdash C^k$, then C occurs at least k times in Δ.*

Proof. If $k = 0$, the result is trivial. Otherwise, we proceed by a simple induction on the derivation of $\Delta \vdash C^k$. □

Lemma 3.1.8. *Let Δ be well-formed. The following implications hold:*

1. *if $\Delta = \Delta', F$, then Δ' is well-formed and $rk(\Delta) \leq rk(\Delta')$;*
2. *if $\Delta = \Delta', !F$, then $\Delta, !F$ is well-formed and $rk(\Delta) = rk(\Delta, !F)$;*
3. *if $\Delta = \Delta', F_1 \otimes F_2$, then Δ', F_1, F_2 is well-formed and $rk(\Delta) = rk(\Delta', F_1, F_2)$;*
4. *if $\Delta = \Delta_1, \Delta_2, F_1 \multimap F_2$, then Δ_2, F_2 is well-formed and $rk(\Delta) \leq rk(\Delta_2, F_2)$;*

5. *if $\Delta = \Delta', \forall x.F$, then for every t we have that $\Delta', F\{t/x\}$ is well-formed and $rk(\Delta) = rk(\Delta', F\{t/x\})$;*

6. *if $\Delta = \Delta', !F$, then Δ', F is well-formed and $rk(\Delta) = rk(\Delta', F)$.*

Proof. By simple syntactic checks. $\square$

Lemma 3.1.9. *Let Δ be weakly guarded. The following implications hold:*

1. *if $\Delta = \Delta', F$, then Δ' is weakly guarded;*

2. *if $\Delta = \Delta', !F$, then $\Delta, !F$ is weakly guarded;*

3. *if $\Delta = \Delta', F_1 \otimes F_2$, then Δ', F_1, F_2 is weakly guarded;*

4. *if $\Delta = \Delta_1, \Delta_2, F_1 \multimap F_2$ and $\Delta_1 \vdash F_1$, then Δ_2, F_2 is weakly guarded;*

5. *if $\Delta = \Delta', \forall x.F$, then $\Delta', F\{t/x\}$ is weakly guarded for every t;*

6. *if $\Delta = \Delta', !F$, then Δ', F is weakly guarded.*

Proof. For all the points of the statement, let Δ_c denote the multiset in the conclusion. Lemma 3.1.8 guarantees that Δ_c is well-formed. Now we observe that in each case, for every formula F, we have that $\Delta_c \vdash F$ implies $\Delta \vdash F$ by the application of a specific rule of the logic. Thus, let us assume by contradiction that $\Delta_c \vdash C^n$ with $n > 1$ for some active control formula C. By the previous observation, we have $\Delta \vdash C^n$, but this is contradictory, since Δ is weakly guarded by hypothesis. $\square$

Lemma 3.1.10 (Stratification)**.** *Let $\Delta = \Delta', F_1, \ldots, F_m$ be well-formed and let C be active in Δ. If $\Delta \vdash C^n$ with $n \geq 1$ and $\forall i \in [1, m] : rk(F_i) > rk(C)$, then $\Delta' \vdash C^n$.*

Proof. By induction on the derivation of $\Delta \vdash C^n$:

Case (IDENT): the case is trivial, since the hypothesis on the rank cannot hold;

Case (WEAK): let us assume that the principal formula is F_1, so we have $\Delta \vdash C^n$ by the premise $\Delta', F_2, \ldots, F_m \vdash C^n$. The latter multiset is well-formed and C is active there by Lemma 3.1.8, so we can apply the inductive hypothesis to get $\Delta' \vdash C^n$. Otherwise, let the principal formula belong to Δ', the conclusion follows by inductive hypothesis and (WEAK);

Case (CONTR): let us assume that the principal formula is F_1, so we have $\Delta \vdash C^n$ by the premise $\Delta, F_1 \vdash C^n$. The latter multiset is well-formed and C is active there by Lemma 3.1.8, so we can apply the inductive hypothesis to get $\Delta' \vdash C^n$. Otherwise, let the principal formula belong to Δ', the conclusion follows by inductive hypothesis and (CONTR);

Case ($\otimes$-LEFT): let us assume that the principal formula is $F_1 = F' \otimes F''$, so we have $\Delta \vdash C^n$ by the premises $\Delta', F', F'', F_2, \ldots, F_m \vdash C^n$. The latter multiset is well-formed and C is active there by Lemma 3.1.8 . Since $rk(F' \otimes F'') = min\ \{rk(F'), rk(F'')\}$ and $rk(F' \otimes F'') > rk(C)$, we know that both $rk(F') > rk(C)$ and $rk(F'') > rk(C)$. By inductive hypothesis we then get $\Delta' \vdash C^n$ as desired. Otherwise, let the principal formula belong to Δ', the conclusion follows by inductive hypothesis and ($\otimes$-LEFT);

Case ($\otimes$-RIGHT): let $\Delta \vdash C^n$ with $n \geq 2$ by the premises $\Delta_1 \vdash C^j$ and $\Delta_2 \vdash C^k$ with $\Delta = \Delta_1, \Delta_2$ and $j + k = n$. Let us assume without loss of generality that $\Delta_1 = \Delta'_1, F_1, \ldots, F_h$ and $\Delta_2 = \Delta'_2, F_{h+1}, \ldots, F_m$. Both Δ_1 and Δ_2 are well-formed and C is active there by Lemma 3.1.8. By inductive hypothesis we get $\Delta'_1 \vdash C^j$ and $\Delta'_2 \vdash C^k$, so we conclude $\Delta' \vdash C^n$ by an application of ($\otimes$-RIGHT);

Case ($\multimap$-LEFT): let us assume that the principal formula is $F_1 = F' \multimap F''$. Since Δ is well-formed, we can distinguish three cases:

- $F' = B_1, F'' = B_2$. We have $\Delta_1 \vdash B_1$ and $\Delta_2, B_2 \vdash C^n$ with $\Delta', F_2, \ldots, F_m = \Delta_1, \Delta_2$. Let us assume without loss of generality that $\Delta_1 = \Delta'_1, F_2, \ldots, F_h$ and $\Delta_2 = \Delta'_2, F_{h+1}, \ldots, F_m$. We know that Δ_2, B_2 is well-formed and C is active there by Lemma 3.1.8. Moreover, we note that $rk(B_2) = +\infty$, while $rk(C)$ is finite, so we can apply the inductive hypothesis to get $\Delta'_2 \vdash C^n$. The conclusion follows by applying (WEAK) an appropriate number of times;
- $F' = \hat{C}, F'' = \hat{P}$. We have $\Delta_1 \vdash \hat{C}$ and $\Delta_2, \hat{P} \vdash C^n$ with $\Delta', F_2, \ldots, F_m = \Delta_1, \Delta_2$. Let us assume without loss of generality that $\Delta_1 = \Delta'_1, F_2, \ldots, F_h$ and $\Delta_2 = \Delta'_2, F_{h+1}, \ldots, F_m$. We know that $\Delta_2, \hat{P}$ is well-formed and C is active there by Lemma 3.1.8. Moreover, we note that $rk(C) \leq rk(\hat{C}) < rk(\hat{P})$ by the hypothesis of stratification, so we can apply the inductive hypothesis to get $\Delta'_2 \vdash C^n$. The conclusion follows by applying (WEAK) an appropriate number of times;
- $F' = \hat{P}, F'' = !(\hat{C} \multimap \hat{P})$. We have $\Delta_1 \vdash \hat{P}$ and $\Delta_2, !(\hat{C} \multimap \hat{P}) \vdash C^n$ with $\Delta', F_2, \ldots, F_m = \Delta_1, \Delta_2$. Let us assume without loss of generality that $\Delta_1 = \Delta'_1, F_2, \ldots, F_h$ and $\Delta_2 = \Delta'_2, F_{h+1}, \ldots, F_m$. We know that $\Delta_2, !(\hat{C} \multimap \hat{P})$ is well-formed and C is active there by Lemma 3.1.8. Moreover, we note that $rk(!(\hat{C} \multimap \hat{P})) = +\infty$, while $rk(C)$ is finite, so we can apply the inductive hypothesis to get $\Delta'_2 \vdash C^n$. The conclusion follows by applying (WEAK) an appropriate number of times.

Otherwise, let the principal formula be $F = F' \multimap F''$ with $\Delta' = \Delta'', F$. Without loss of generality, we can assume $rk(F) \leq rk(C)$, but this is contradictory, since any implication has an infinite rank, which is strictly greater than the finite rank of C.

Case ($\forall$-LEFT): let us assume that the principal formula is $F_1 = \forall x.F'$, so we have $\Delta \vdash C^n$ by the premises $\Delta', F'\{t/x\}, F_2, \ldots, F_m \vdash C^n$ for some t. The latter multiset is well-formed and C is active there by Lemma 3.1.8. Note that $rk(\forall x.F') = rk(F'\{t/x\}) = +\infty$, since the multiset is well-formed, hence by inductive hypothesis we get $\Delta' \vdash C^n$ as desired. Otherwise, let the principal formula belong to Δ', the conclusion follows by inductive hypothesis and ($\forall$-LEFT);

Case (!-LEFT): let us assume that the principal formula is $F_1 = !F'$, so we have $\Delta \vdash C^n$ by the premises $\Delta', F', F_2, \ldots, F_m \vdash C^n$. The latter multiset is well-formed and C is active there by Lemma 3.1.8. Note that $rk(!F') = rk(F') = +\infty$, since the multiset is well-formed, hence by inductive hypothesis we get $\Delta' \vdash C^n$ as desired. Otherwise, let the principal formula belong to Δ', the conclusion follows by inductive hypothesis and (!-LEFT).

□

Corollary 3.1.11. *Let Δ be well-formed. If $\Delta \vdash C$ and C is active in Δ, then Δ contains at least an affine formula.*

Proof. Let Δ' be the multiset obtained from Δ by removing all the formulas F such that $rk(F) > rk(C)$. By Lemma 3.1.10 we have $\Delta' \vdash C$. Clearly, Δ' must contain at least a formula $\hat{F}$, since $\emptyset \nvdash C$, and by construction we know that $rk(\hat{F}) \leq rk(C)$. Since $rk(C)$ is finite, also $rk(\hat{F})$ is finite, so this formula must be affine by Definition 2.4.1. □

Lemma 3.1.12. *Let Δ be weakly guarded, then Δ, B is weakly guarded. Moreover, we have $rk(\Delta) = rk(\Delta, B)$.*

Proof. It is immediate to note that Δ, B is well-formed and that the introduction of B does not change the rank of the multiset. As to weak guardedness, let us assume by contradiction that $\Delta, B \vdash C^n$ with $n > 1$ for some active control formula C. Since $rk(C)$ is finite, while $rk(B) = +\infty$, we have that $\Delta \vdash C^n$ by Lemma 3.1.10. But this is contradictory, since Δ is weakly guarded. □

Lemma 3.1.13 (Strengthening)**.** *Let $\Delta = \Delta', C \multimap P$ be well-formed and let C be active in Δ. If $\Delta \vdash P'$ and $\Delta' \nvdash C$, then $\Delta' \vdash P'$.*

Proof. By induction on the derivation of $\Delta \vdash P'$. We show just the most interesting cases:

Case (IDENT): the rule cannot be applied, since $C \multimap P$ is not a payload formula;

Case ($\otimes$-LEFT): let $\Delta' = \Delta'', F_1 \otimes F_2$ and let $\Delta \vdash P'$ by $\Delta'', F_1, F_2, C \multimap P \vdash P'$. By Lemma 3.1.8 we know that the latter multiset is well-formed and C is active. Moreover, we note that $\Delta'', F_1, F_2 \nvdash C$, otherwise we could get $\Delta' \vdash C$ by an application of ($\otimes$-LEFT). Thus, we can apply the inductive hypothesis to get $\Delta'', F_1, F_2 \vdash P'$ and conclude by ($\otimes$-LEFT);

Case ($\otimes$-Right): let $\Delta \vdash P_1 \otimes P_2$ by the hypotheses $\Delta_1 \vdash P_1$ and $\Delta_2 \vdash P_2$ with $\Delta = \Delta_1, \Delta_2$. By Lemma 3.1.8 both Δ_1 and Δ_2 are well-formed and C is active in both. Without loss of generality, let us assume $\Delta_1 = \Delta_1', C \multimap P$. We note that $\Delta_1' \nvdash C$, otherwise we could get $\Delta' \vdash C$ by (Weak). Thus, we can apply the inductive hypothesis to get $\Delta_1' \vdash P_1$ and conclude by ($\otimes$-Right);

Case ($\multimap$-Left): we distinguish two cases, according to the principal formula:

- let $\Delta' = \Delta'', F_1 \multimap F_2$ and let $\Delta \vdash P'$ by the hypotheses $\Delta_1 \vdash F_1$ and $\Delta_2, F_2 \vdash P'$ with $\Delta'', C \multimap P = \Delta_1, \Delta_2$. By Lemma 3.1.8 both Δ_1 and Δ_2, F_2 are well-formed and C is active in both. We distinguish two cases, according to whether $C \multimap P$ belongs to Δ_1 or to Δ_2.

  Let $\Delta_1 = \Delta_1', C \multimap P$, we have that F_1 is a payload formula. Moreover, since $\Delta' \nvdash C$ and $\Delta_1' \subseteq \Delta'$, we know that $\Delta_1' \nvdash C$, otherwise we could deduce $\Delta' \vdash C$ by (Weak). Thus, we can apply the inductive hypothesis to get $\Delta_1' \vdash F_1$ and conclude by ($\multimap$-Left).

  Otherwise, let $\Delta_2 = \Delta_2', C \multimap P$. We know that $\Delta_2', F_2 \nvdash C$, otherwise we could deduce $\Delta' \vdash C$ by ($\multimap$-Left). Thus, we can apply the inductive hypothesis to get $\Delta_2', F_2 \vdash P'$ and conclude by ($\multimap$-Left);
- let $\Delta' \vdash P'$ by the hypotheses $\Delta_1 \vdash C$ and $\Delta_2, P \vdash P'$ with $\Delta' = \Delta_1, \Delta_2$. We have a contradiction, since $\Delta_1 \vdash C$ implies $\Delta' \vdash C$ by (Weak).

Case ($\multimap$-Right): the only possibility is that $\Delta \vdash B_1 \multimap B_2$ by the hypothesis $\Delta, B_1 \vdash B_2$. It is immediate to note that Δ, B_1 is well-formed and that C is active, since the introduction of B_1 cannot change the rank of the multiset. Let us assume by contradiction that $\Delta', B_1 \vdash C$. Since $rk(C)$ is finite, while $rk(B_1) = +\infty$, we have that $\Delta' \vdash C$ by Lemma 3.1.10, but this is contradictory. Thus, we have $\Delta', B_1 \nvdash C$, so we can apply the inductive hypothesis to get $\Delta', B_1 \vdash B_2$ and conclude by ($\multimap$-Right);

Case (!-Left): let $\Delta' = \Delta'', !F$ and let $\Delta \vdash P'$ by the hypothesis $\Delta'', F, C \multimap P \vdash P'$. By Lemma 3.1.8 we know that the latter multiset is well-formed and C is active. Moreover, we note that $\Delta'', F \nvdash C$, otherwise we could get $\Delta' \vdash C$ by (!-Left). Thus, we can apply the inductive hypothesis to get $\Delta'', F \vdash P'$ and conclude by (!-Left);

Case (!-Right): the rule cannot be applied, since $\Delta', C \multimap P$ is not exponential.

□

Lemma 3.1.14. *Let $\Delta = \Delta', (C \multimap P)^n$ be weakly guarded. If $\Delta \vdash P'$ and C is active in Δ, then $\Delta', C \multimap P \vdash P'$.*

Proof. By induction on the derivation of $\Delta \vdash P'$. Without loss of generality in the following inductive cases we assume $n \geq 2$, since the conclusion follows by (Weak) for $n = 0$ and it is trivial for $n = 1$. We show just the most interesting cases:

Case (IDENT): the rule cannot be applied, since we are assuming to have at least two copies of $C \multimap P$ in the multiset. Moreover, $C \multimap P$ is not a payload formula;

Case ($\otimes$-LEFT): let $\Delta' = \Delta'', F_1 \otimes F_2$ and let $\Delta \vdash P'$ from $\Delta'', F_1, F_2, (C \multimap P)^n \vdash P'$. The latter multiset is weakly guarded by Lemma 3.1.9 and C is active by Lemma 3.1.8, so we can apply the inductive hypothesis to get $\Delta'', F_1, F_2, C \multimap P \vdash P'$ and conclude $\Delta', C \multimap P \vdash P'$ by ($\otimes$-LEFT);

Case ($\otimes$-RIGHT): let $\Delta \vdash P_1 \otimes P_2$ by the premises $\Delta_1 \vdash P_1$ and $\Delta_2 \vdash P_2$ with $\Delta = \Delta_1, \Delta_2$. Let $\Delta_1 = \Delta_1', (C \multimap P)^h$ and $\Delta_2 = \Delta_2', (C \multimap P)^k$ with $h + k = n$. Both Δ_1 and Δ_2 are weakly guarded by Lemma 3.1.9 and C is active in both by Lemma 3.1.8. We apply the inductive hypothesis to both $\Delta_1 \vdash P_1$ and $\Delta_2 \vdash P_2$ to get $\Delta_1', C \multimap P \vdash P_1$ and $\Delta_2', C \multimap P \vdash P_2$. Now we assume by contradiction that both $\Delta_1' \vdash C$ and $\Delta_2' \vdash C$, then we have $\Delta \vdash C \otimes C$ by ($\otimes$-RIGHT) and an appropriate number of applications of (WEAK), but, given that Δ is weakly guarded, this is contradictory. Without loss of generality we can then assume that $\Delta_1' \nvdash C$, so by Lemma 3.1.13 we have $\Delta_1' \vdash P_1$ and we conclude $\Delta_1', \Delta_2', C \multimap P \vdash P_1 \otimes P_2$ by ($\otimes$-RIGHT);

Case ($\multimap$-LEFT): we distinguish four cases, according to the principal formula:

- let $\Delta \vdash P'$ by the premises $\Delta_1 \vdash C$ and $\Delta_2, P \vdash P'$ with $\Delta', (C \multimap P)^{n-1} = \Delta_1, \Delta_2$. Let $\Delta_1 = \Delta_1', (C \multimap P)^h$ and $\Delta_2 = \Delta_2', (C \multimap P)^k$ with $h + k = n - 1$. By Lemma 3.1.9 we have that both Δ_1 and Δ_2, P are weakly guarded and C is active in both by Lemma 3.1.8. Moreover, C is a control formula, i.e., it is a payload formula. We are thus allowed to apply the inductive hypothesis to both $\Delta_1 \vdash C$ and $\Delta_2, P \vdash P'$ to get $\Delta_1', C \multimap P \vdash C$ and $\Delta_2', P, C \multimap P \vdash P'$. Since $\mathit{rk}(C \multimap P) = +\infty > \mathit{rk}(C)$, by Lemma 3.1.10 we have $\Delta_1' \vdash C$. Let us assume by contradiction that $\Delta_2', P \vdash C$: since $\mathit{rk}(C) < \mathit{rk}(P)$ by the stratification hypothesis, by Lemma 3.1.10 we know that $\Delta_2' \vdash C$, but this is contradictory, since we would first get $\Delta_1', \Delta_2' \vdash C \otimes C$ by ($\otimes$-RIGHT) and then $\Delta \vdash C \otimes C$ by (WEAK). Thus, $\Delta_2', P \nvdash C$ and we have $\Delta_2', P \vdash P'$ by Lemma 3.1.13. We can then conclude $\Delta_1', \Delta_2', C \multimap P \vdash P'$ by an application of ($\multimap$-LEFT);
- let $\Delta' = \Delta'', B_1 \multimap B_2$ by the premises $\Delta_1 \vdash B_1$ and $\Delta_2, B_2 \vdash P'$ with $\Delta'', (C \multimap P)^n = \Delta_1, \Delta_2$. Let $\Delta_1 = \Delta_1', (C \multimap P)^h$ and $\Delta_2 = \Delta_2', (C \multimap P)^k$ with $h + k = n$. By Lemma 3.1.9 we have that both Δ_1 and Δ_2, B_2 are weakly guarded and C is active in both by Lemma 3.1.8. Moreover, B_1 is a base formula, i.e., it is a payload formula. We are thus allowed to apply the inductive hypothesis to both $\Delta_1 \vdash B_1$ and $\Delta_2, B_2 \vdash P'$ to get $\Delta_1', C \multimap P \vdash B_1$ and $\Delta_2', B_2, C \multimap P \vdash P'$. Let us assume by contradiction that both $\Delta_1' \vdash C$ and $\Delta_2', B_2 \vdash C$: since $\mathit{rk}(B_2) = +\infty > \mathit{rk}(C)$, by Lemma 3.1.10 we have $\Delta_2' \vdash C$, whence $\Delta \vdash C \otimes C$ by ($\otimes$-RIGHT) and (WEAK), which is contradictory. Thus, we can apply Lemma 3.1.13 either on $\Delta_1', C \multimap P \vdash B_1$

or on $\Delta'_2, B_2, C \multimap P \vdash P'$ to remove $C \multimap P$ from the judgement. The conclusion follows by ($\multimap$-LEFT);

- let $\Delta' = \Delta'', \hat{C} \multimap \hat{P}$ by the premises $\Delta_1 \vdash \hat{C}$ and $\Delta_2, \hat{P} \vdash P'$ with $\Delta'', (C \multimap P)^n = \Delta_1, \Delta_2$. Let $\Delta_1 = \Delta'_1, (C \multimap P)^h$ and $\Delta_2 = \Delta'_2, (C \multimap P)^k$ with $h+k = n$. By Lemma 3.1.9 we have that both Δ_1 and $\Delta_2, \hat{P}$ are weakly guarded and C is active in both by Lemma 3.1.8. Moreover, $\hat{C}$ is a control formula, i.e., it is a payload formula. We are thus allowed to apply the inductive hypothesis to both $\Delta_1 \vdash \hat{C}$ and $\Delta_2, \hat{P} \vdash P'$ to get $\Delta'_1, C \multimap P \vdash \hat{C}$ and $\Delta'_2, \hat{P}, C \multimap P \vdash P'$. Let us assume by contradiction that both $\Delta'_1 \vdash C$ and $\Delta'_2, \hat{P} \vdash C$: since $rk(C) \leq rk(\hat{C}) < rk(\hat{P})$ by the stratification hypothesis, by Lemma 3.1.10 we have $\Delta'_2 \vdash C$, whence $\Delta \vdash C \otimes C$ by ($\otimes$-RIGHT) and (WEAK), which is contradictory. Thus, we can apply Lemma 3.1.13 either on $\Delta'_1, C \multimap P \vdash \hat{C}$ or on $\Delta'_2, \hat{P}, C \multimap P \vdash P'$ to remove $C \multimap P$ from the judgement. The conclusion follows by ($\multimap$-LEFT);
- let $\Delta' = \Delta'', \hat{P} \multimap !(\hat{C} \multimap \hat{P})$ by the premises $\Delta_1 \vdash \hat{P}$ and $\Delta_2, !(\hat{C} \multimap \hat{P}) \vdash P'$ with $\Delta'', (C \multimap P)^n = \Delta_1, \Delta_2$. Let $\Delta_1 = \Delta'_1, (C \multimap P)^h$ and $\Delta_2 = \Delta'_2, (C \multimap P)^k$ with $h+k = n$. By Lemma 3.1.9 we have that both Δ_1 and $\Delta_2, !(\hat{C} \multimap \hat{P})$ are weakly guarded and C is active in both by Lemma 3.1.8. Moreover, $\hat{P}$ is a payload formula. We are thus allowed to apply the inductive hypothesis to both $\Delta_1 \vdash \hat{P}$ and $\Delta_2, !(\hat{C} \multimap \hat{P}) \vdash P'$ to get $\Delta'_1, C \multimap P \vdash \hat{P}$ and $\Delta'_2, !(\hat{C} \multimap \hat{P}), C \multimap P \vdash P'$. Let us assume by contradiction that both $\Delta'_1 \vdash C$ and $\Delta'_2, !(\hat{C} \multimap \hat{P}) \vdash C$: since $rk(!(\hat{C} \multimap \hat{P})) = +\infty > rk(C)$, by Lemma 3.1.10 we have $\Delta'_2 \vdash C$, whence $\Delta \vdash C \otimes C$ by ($\otimes$-RIGHT) and (WEAK), which is contradictory. Thus, we can apply Lemma 3.1.13 either on $\Delta'_1, C \multimap P \vdash \hat{P}$ or on $\Delta'_2, !(\hat{C} \multimap \hat{P}), C \multimap P \vdash P'$ to remove $C \multimap P$ from the judgement. The conclusion follows by ($\multimap$-LEFT);

Case ($\multimap$-RIGHT): let $\Delta \vdash B_1 \multimap B_2$ by the hypothesis $\Delta, B_1 \vdash B_2$. By Lemma 3.1.12 we know that Δ, B_1 is weakly guarded and C is active, so by inductive hypothesis we get $\Delta', B_1, C \multimap P \vdash B_2$ and we conclude by ($\multimap$-RIGHT);

Case (!-LEFT): let $\Delta' = \Delta'', !F$ and let $\Delta \vdash P'$ by the hypothesis $\Delta'', F, (C \multimap P)^n \vdash P'$. The latter multiset is weakly guarded by Lemma 3.1.9 and C is active by Lemma 3.1.8, so we can apply the inductive hypothesis to get $\Delta'', F, C \multimap P \vdash P'$ and we conclude by (!-LEFT);

Case (!-RIGHT): the rule cannot be applied, since we are assuming to have at least two copies of $C \multimap P$ in the multiset, i.e., the multiset is not exponential.

□

Lemma 3.1.15 (Dereliction). *Let $\Delta = \Delta', !(C \multimap P)$ be weakly guarded. If $\Delta \vdash P'$ and C is active in Δ, then $\Delta', C \multimap P \vdash P'$.*

Proof. We prove a stronger statement, namely:

$$\forall n \geq 0 : \Delta', (!(C \multimap P))^n \vdash P' \text{ implies } \Delta', C \multimap P \vdash P',$$

provided that the initial hypotheses are satisfied. We proceed by induction on the derivation of the judgement in the premise, we show just the most interesting cases:

Case (Ident): if $n > 0$, this rule cannot be applied, since $!(C \multimap P)$ is not a payload formula. If $n = 0$, we have $P' \vdash P'$ by hypothesis and we conclude $P', C \multimap P \vdash P'$ by (Weak);

Case (Weak): we have two cases. If the principal formula is $!(C \multimap P)$, the conclusion is immediate by inductive hypothesis. Otherwise, if the principal formula belongs to Δ', the conclusion follows by applying the inductive hypothesis and (Weak);

Case (Contr): we have two cases. If the principal formula is $!(C \multimap P)$, the conclusion is immediate by inductive hypothesis. Otherwise, if the principal formula belongs to Δ', the conclusion follows by applying the inductive hypothesis and (Contr);

Case ($\otimes$-Left): $\Delta' = \Delta'', F_1 \otimes F_2$ and let $\Delta \vdash P'$ from $\Delta'', F_1, F_2, (!(C \multimap P))^n \vdash P'$. The latter multiset is weakly guarded by Lemma 3.1.9 and C is active by Lemma 3.1.8, so we can apply the inductive hypothesis to get $\Delta'', F_1, F_2, C \multimap P \vdash P'$ and conclude $\Delta', C \multimap P \vdash P'$ by ($\otimes$-Left);

Case ($\otimes$-Right): let $\Delta \vdash P_1 \otimes P_2$ by the hypotheses $\Delta_1 \vdash P_1$ and $\Delta_2 \vdash P_2$ with $\Delta = \Delta_1, \Delta_2$. Let $\Delta_1 = \Delta'_1, (!(C \multimap P))^h$ and $\Delta_2 = \Delta'_2, (!(C \multimap P))^k$ with $h+k = n$. We have that both Δ_1 and Δ_2 are weakly guarded by Lemma 3.1.9 and C is active in both by Lemma 3.1.8. We apply the inductive hypothesis to both $\Delta_1 \vdash P_1$ and $\Delta_2 \vdash P_2$ to get $\Delta'_1, C \multimap P \vdash P_1$ and $\Delta'_2, C \multimap P \vdash P_2$. Thus, we have $\Delta'_1, \Delta'_2, C \multimap P, C \multimap P \vdash P_1 \otimes P_2$ by ($\otimes$-Right) and we conclude $\Delta'_1, \Delta'_2, C \multimap P \vdash P_1 \otimes P_2$ by Lemma 3.1.14;

Case ($\multimap$-Left): let $\Delta' = \Delta'', F_1 \multimap F_2$ and let $\Delta \vdash P'$ by the hypotheses $\Delta_1 \vdash F_1$ and $\Delta_2, F_2 \vdash P'$ with $\Delta'', (!(C \multimap P))^n = \Delta_1, \Delta_2$. Let $\Delta_1 = \Delta'_1, (!(C \multimap P))^h$ and $\Delta_2 = \Delta'_2, (!(C \multimap P))^k$ with $h + k = n$. By Lemma 3.1.9 we have that both Δ_1 and Δ_2, F_2 are weakly guarded and C is active in both by Lemma 3.1.8. Moreover, we note that F_1 is a payload formula. We are thus allowed to apply the inductive hypothesis to both $\Delta_1 \vdash F_1$ and $\Delta_2, F_2 \vdash P'$ to get respectively $\Delta'_1, C \multimap P \vdash F_1$ and $\Delta'_2, F_2, C \multimap P \vdash P'$. Thus, we get $\Delta', C \multimap P, C \multimap P \vdash P'$ by ($\multimap$-Left) and we conclude $\Delta', C \multimap P \vdash P'$ by Lemma 3.1.14;

Case ($\multimap$-Right): let $\Delta \vdash B_1 \multimap B_2$ by the hypothesis $\Delta, B_1 \vdash B_2$. By Lemma 3.1.12 we know that Δ, B_1 is weakly guarded and C is active, so by inductive hypothesis we get $\Delta', B_1, C \multimap P \vdash B_2$ and we conclude by ($\multimap$-Right);

Case (!-Left): if the principal formula is $!(C \multimap P)$, we have $\Delta', C \multimap P, !(C \multimap P)^{n-1} \vdash P'$. If $n-1=0$, we are done, otherwise we apply the inductive hypothesis to get $\Delta', C \multimap P, C \multimap P \vdash P'$ and we conclude by Lemma 3.1.14. Otherwise, if the principal formula belongs to Δ', the conclusion follows by first applying the inductive hypothesis and then using (!-Left);

Case (!-Right): we have $\Delta', (!(C \multimap P))^n \vdash !B$ by the hypothesis $\Delta', (!(C \multimap P))^n \vdash B$ with Δ' exponential. By inductive hypothesis $\Delta', C \multimap P \vdash B$. Let us assume by contradiction that $\Delta' \vdash C$: then, by Corollary 3.1.11, there exists an affine formula in Δ', but this is contradictory. Thus, we have $\Delta' \nvdash C$, which implies $\Delta' \vdash B$ by Lemma 3.1.13. We can then apply (!-Right) to derive $\Delta' \vdash !B$ and we conclude $\Delta', C \multimap P \vdash !B$ by (Weak).

□

Lemma 3.1.16. *Let $\Delta = \Delta', C \multimap P$ be well-formed. If $\Delta \vdash P'$, then $\Delta', P \vdash P'$.*

Proof. By induction on the derivation of $\Delta \vdash P'$. The proof strongly resembles those of the previous results, but it is actually easier. We just show the most interesting cases:

Case (Ident): the rule cannot be applied, since $C \multimap P$ is not a payload formula;

Case (Weak): if the principal formula is $C \multimap P$, the conclusion follows by (Weak). Otherwise, if the principal formula belongs to Δ', the conclusion follows by first applying the inductive hypothesis and then using (Weak);

Case ($\otimes$-Left): let $\Delta' = \Delta'', F_1 \otimes F_2$ and let $\Delta \vdash P'$ from $\Delta'', F_1, F_2, C \multimap P \vdash P'$. The latter multiset is well-formed by Lemma 3.1.8, so we can apply the inductive hypothesis to get $\Delta'', F_1, F_2, P \vdash P'$ and conclude $\Delta', P \vdash P'$ by ($\otimes$-Left);

Case ($\otimes$-Right): let $\Delta \vdash P_1 \otimes P_2$ by the hypotheses $\Delta_1 \vdash P_1$ and $\Delta_2 \vdash P_2$ with $\Delta = \Delta_1, \Delta_2$. By Lemma 3.1.8 both Δ_1 and Δ_2 are well-formed, so we are allowed to apply the inductive hypothesis to either $\Delta_1 \vdash P_1$ or $\Delta_2 \vdash P_2$, according to whether $C \multimap P$ occurs in Δ_1 or in Δ_2, and conclude by ($\otimes$-Right);

Case ($\multimap$-Left): we distinguish two cases, according to the principal formula:

- let $\Delta' = \Delta'', F_1 \multimap F_2$ and let $\Delta \vdash P'$ by the hypotheses $\Delta_1 \vdash F_1$ and $\Delta_2, F_2 \vdash P'$ with $\Delta'', C \multimap P = \Delta_1, \Delta_2$. By Lemma 3.1.8 we have that both Δ_1 and Δ_2, F_2 are well-formed. Moreover, we note that F_1 is a payload formula. We are thus allowed to apply the inductive hypothesis to either $\Delta_1 \vdash F_1$ or $\Delta_2, F_2 \vdash P'$, according to whether $C \multimap P$ occurs in Δ_1 or in Δ_2, and conclude by ($\multimap$-Left);
- let $\Delta \vdash P'$ by the hypotheses $\Delta_1 \vdash C$ and $\Delta_2, P \vdash P'$ with $\Delta' = \Delta_1, \Delta_2$. We conclude $\Delta', P \vdash P'$ by applying (Weak) for an appropriate number of times.

Case ($\multimap$-RIGHT): let $\Delta \vdash B_1 \multimap B_2$ by the hypothesis $\Delta, B_1 \vdash B_2$. It is immediate to note that Δ, B_1 is well-formed, so by inductive hypothesis we get $\Delta', B_1, P \vdash B_2$ and we conclude $\Delta', P \vdash B_1 \multimap B_2$ by ($\multimap$-RIGHT);

Case (!-LEFT): let $\Delta' = \Delta'', !F$ and let $\Delta \vdash P'$ by the hypothesis $\Delta'', F, C \multimap P \vdash P'$. The latter multiset is well-formed by Lemma 3.1.8, so we can apply the inductive hypothesis to get $\Delta'', F, P \vdash P'$ and we conclude $\Delta', P \vdash P'$ by (!-LEFT);

Case (!-RIGHT): the rule cannot be applied, since $C \multimap P$ is not exponential.

□

Corollary 3.1.17 (Bounded Usage)**.** *Let $\Delta = \Delta', !(C \multimap P)$ be weakly guarded. If $\Delta \vdash P'$ and C is active in Δ, then $\Delta', P \vdash P'$.*

Proof. By Lemma 3.1.15 we have $\Delta', C \multimap P \vdash P'$. By Lemma 3.1.9 we know that $\Delta', C \multimap P$ is weakly guarded, i.e., it is well-formed, thus the conclusion follows by Lemma 3.1.16. □

3.2 Soundness of the type system

3.2.1 Properties of the logic

Lemma 3.2.1 (Substitution for the Logic)**.** *For all Δ, F and all substitutions σ of variables with closed terms, it holds that $\Delta \vdash F$ implies $\Delta\sigma \vdash F\sigma$.*

Proof. By induction on the derivation of $\Delta \vdash F$.

Case (IDENT): we can immediately conclude by (IDENT).

Case ($\forall$-RIGHT): we know that $F = \forall x.F'$ and:

$$\frac{\Delta \vdash F' \qquad x \notin fv(\Delta)}{\Delta \vdash \forall x.F'}$$

We define the slightly modified substitution σ' as follows:

$$y\sigma' := \begin{cases} y\sigma & \text{if } x \neq y \\ y & \text{if } x = y \end{cases}$$

It follows that $(\forall x.F')\sigma = \forall x.(F'\sigma')$. Since $x \notin fv(\Delta)$, we know that $\Delta\sigma = \Delta\sigma'$. We apply the induction hypothesis to $\Delta \vdash F'$ and σ', so we get $\Delta\sigma' \vdash F'\sigma'$. Using the previous observations, we conclude $\Delta\sigma \vdash (\forall x.F')\sigma$ by an application of ($\forall$-RIGHT). Notice that the rule can be applied, since $x \notin fv(\Delta\sigma)$ by the assumption that σ does not introduce variables.

Case ($\forall$-Left): we know that $\Delta \triangleq \Delta', \forall x.F'$ and:

$$\frac{\Delta', F'\{t/x\} \vdash F}{\Delta', \forall x.F' \vdash F}$$

We define the slightly modified substitution σ' as follows:

$$y\sigma' := \begin{cases} y\sigma & \text{if } x \neq y \\ y & \text{if } x = y \end{cases}$$

By the induction hypothesis we know that $(\Delta', F'\{t/x\})\sigma \vdash F\sigma$. This is equivalent to $\Delta'\sigma, (F'\{t/x\})\sigma \vdash F\sigma$, which is equivalent to $\Delta'\sigma, (F'\sigma')\{t\sigma/x\} \vdash F\sigma$ by the definition of σ and σ' and the fact that both σ and σ' do not introduce variables. We can apply ($\forall$-Left) to derive:

$$\frac{\Delta'\sigma, (F'\sigma')\{t\sigma/x\} \vdash F\sigma}{\Delta'\sigma, \forall x.(F'\sigma') \vdash F\sigma}$$

We know that by definition of σ, σ' it holds that $\Delta'\sigma, \forall x.(F'\sigma') \vdash F\sigma$ is equivalent to $\Delta'\sigma, (\forall x.F')\sigma \vdash F\sigma$ and thus to $(\Delta', \forall x.F')\sigma \vdash F\sigma$, which is the conclusion.

Case (=-Subst): we know that $\Delta \triangleq \Delta', t = t'$ and:

$$\frac{\exists\sigma' = mgu(t, t') \Rightarrow \Delta'\sigma' \vdash F\sigma'}{\Delta', t = t' \vdash F}$$

We need to show that $(\Delta', t = t')\sigma \vdash F\sigma$, which by definition of substitution is equivalent to showing that $\Delta'\sigma, t\sigma = t'\sigma \vdash F\sigma$. We distinguish two cases: if there does not exist a most general unifier $\sigma'' = mgu(t\sigma, t'\sigma)$, the premise for concluding by an application of (=-Subst) is immediately met and we are done.

Otherwise, we know that there exists $\sigma'' = mgu(t\sigma, t'\sigma)$. By definition of most general unifier, we know that $(t\sigma)\sigma''$ and $(t'\sigma)\sigma''$ are identical, which in particular means that $\sigma \circ \sigma''$ is a unifier for t and t'; this also implies the existence of a most general unifier $\sigma' = mgu(t, t')$ and a (potentially empty) substitution σ''' such that $\sigma \circ \sigma'' = \sigma' \circ \sigma'''$. We can apply the induction hypothesis to $\Delta'\sigma' \vdash F\sigma'$ and σ''' and derive that $(\Delta'\sigma')\sigma''' \vdash (F\sigma')\sigma'''$. As we have seen above, this is equivalent to $(\Delta'\sigma)\sigma'' \vdash (F\sigma)\sigma''$. We can then apply (=-Subst) to conclude that:

$$\frac{\sigma'' = mgu(t\sigma, t'\sigma) \qquad (\Delta'\sigma)\sigma'' \vdash (F\sigma)\sigma''}{\Delta'\sigma, t\sigma = t'\sigma \vdash F\sigma}$$

Case (=-Refl): we can immediately conclude by (=-Refl).

Case (False): we can immediately conclude by (False).

In all other cases we apply the induction hypothesis to the premises of the rule and conclude by applying the rule again. □

Lemma 3.2.2 (Properties of Conjunction)**.** *The following properties hold:*

1. *For all $n \geq 0$, we have $\Delta, F_1, \ldots, F_n \vdash F$ iff $\Delta, F_1 \otimes \ldots \otimes F_n \vdash F$.*
2. *For all Δ, Δ' it holds that $\Delta' \subseteq \Delta$ implies $\Delta \vdash \Delta'$.*

Proof. We proceed as follows:

1. We show both directions separately:
   - $\Delta, F_1 \otimes \ldots \otimes F_n \vdash F \Rightarrow \Delta, F_1, \ldots, F_n \vdash F$: by induction on n.
     - The case for $n = 1$ is trivial.
     - We show the case for $n = 2$ in detail.
       We know that $\Delta, F_1 \otimes F_2 \vdash F$ and need to show that $\Delta, F_1, F_2 \vdash F$.
       We know that:
       $$\dfrac{\dfrac{}{F_1 \vdash F_1}\,\textsc{Ident} \qquad \dfrac{}{F_2 \vdash F_2}\,\textsc{Ident}}{F_1, F_2 \vdash F_1 \otimes F_2}\,\otimes\text{-}\textsc{Right}$$
       Since $F_1, F_2 \vdash F_1 \otimes F_2$ and $\Delta, F_1 \otimes F_2 \vdash F$ we can apply (Cut) to derive that $\Delta, F_1, F_2 \vdash F$.
     - In the remaining cases $n > 2$ we know that $F_1 \otimes \ldots \otimes F_n$ actually denotes a formula of the form $(F_1 \otimes \ldots \otimes F_i) \otimes (F_{i+1} \otimes \ldots \otimes F_n)$, where $F_1 \otimes \ldots \otimes F_i$ and $F_{i+1} \otimes \ldots \otimes F_n$ also contain disambiguating parentheses, for $i \in \{1, \ldots, n-1\}$. We apply the induction hypothesis (for $2 < n$) to the top-level conjunction, which lets us derive that $\Delta, (F_1 \otimes \ldots \otimes F_i), (F_{i+1} \otimes \ldots \otimes F_n) \vdash F$. We then apply the induction hypothesis (for $i < n$) to $F_1 \otimes \ldots \otimes F_i$ and derive that $\Delta, F_1, \ldots, F_i, (F_{i+1} \otimes \ldots \otimes F_n) \vdash F$. Applying the induction hypothesis (for $n - i < n$) to $F_{i+1} \otimes \ldots \otimes F_n$ lets us conclude.
   - $\Delta, F_1, \ldots, F_n \vdash F \Rightarrow \Delta, F_1 \otimes \ldots \otimes F_n \vdash F$: by induction on n.
     The case $n = 1$ is trivial, the case for $n = 2$ follows by ($\otimes$-Left). The remaining cases $n > 2$ follow by applying the induction hypothesis three times, similar to the previous direction.
2. Let $\Delta' = \emptyset$. We interpret $\Delta \vdash \emptyset$ as $\Delta \vdash \mathbf{1}$, which is defined as $\Delta \vdash () = ()$. This immediately follows by (=-Refl).

   Let $\Delta' = F_1, \ldots, F_n$ for some $F_1, \ldots, F_n$. By (Ident) we know that $F_1 \otimes \ldots \otimes F_n \vdash F_1 \otimes \ldots \otimes F_n$. Using property (1) it follows that $F_1, \ldots, F_n \vdash F_1 \otimes \ldots \otimes F_n$, which is equivalent to $\Delta' \vdash \Delta'$ using our standard notation. We can conclude using (Weak) repeatedly.

□

Lemma 3.2.3 (Cut for Sets)**.** *If $\Delta \vdash \Delta'$ and $\Delta', \Delta'' \vdash \Delta'''$, then $\Delta, \Delta'' \vdash \Delta'''$.*

Proof. Let $\Delta' = \emptyset$. We know that $\Delta'' \vdash \Delta'''$ and can immediately conclude by repeated applications of (WEAK). Let then $\Delta' = F_1, \ldots, F_n$ for some $F_1, \ldots, F_n$. We know that $\Delta \vdash F_1, \ldots, F_n$, which denotes $\Delta \vdash F_1 \otimes \ldots \otimes F_n$. Using Lemma 3.2.2 we also know that $F_1 \otimes \ldots \otimes F_n, \Delta'' \vdash \Delta'''$. We can conclude by (CUT). □

Lemma 3.2.4 (Properties of Contraction)**.** *The following properties hold:*

1. *For all Δ it holds that $!\Delta \vdash !\Delta, !\Delta$.*
2. *For all Δ, Δ' it holds that if $\Delta \vdash !\Delta'$, then $\Delta \vdash !\Delta', !\Delta'$.*

Proof. We proceed as follows:

1. We know that $!\Delta, !\Delta \vdash !\Delta, !\Delta$ by Lemma 3.2.2. We can conclude by applying (CONTR) to each element in $!\Delta$.
2. Using property (1) we know that $!\Delta' \vdash !\Delta', !\Delta'$. Since $\Delta \vdash !\Delta'$, we can conclude using Lemma 3.2.3.

□

3.2.2 Basic results

Lemma 3.2.5 (Derived Judgements)**.** *It holds that:*

1. *If $\Gamma; \Delta \vdash \diamond$, then $fnfv(\Delta) \subseteq dom(\Gamma)$ and $\forall \Delta' \subseteq \Delta : \Gamma; \Delta' \vdash \diamond$.*
2. *If $\Gamma; \Delta \vdash \diamond$ and $(x : T) \in \Gamma$, then $T = \psi(T)$.*
3. *If $\Gamma; \Delta \vdash T$, then $\Gamma; \emptyset \vdash \psi(T)$.*
4. *If $\Gamma; \Delta \vdash T$, then $\Gamma; \Delta \vdash \diamond$ and $fnfv(T) \subseteq dom(\Gamma)$.*
5. *If $\Gamma; \Delta \vdash F$, then $\Gamma; \Delta \vdash \diamond$ and $fnfv(F) \subseteq dom(\Gamma)$.*
6. *If $\Gamma; \Delta \hookrightarrow \Gamma; \Delta'$, then $\Gamma; \Delta \vdash \diamond$ and $\Gamma; \Delta' \vdash \diamond$.*
7. *If $\Gamma; \Delta \vdash T :: k$, then $\Gamma; \Delta \vdash T$.*
8. *If $\Gamma; \Delta \vdash T <: T'$, then $\Gamma; \Delta \vdash T$ and $\Gamma; \Delta \vdash T'$.*
9. *If $\Gamma; \Delta \vdash E : T$, then $\Gamma; \Delta \vdash T$ and $fnfv(E) \subseteq dom(\Gamma)$.*

Proof. By induction on the depth of the derivation of the judgements. □

Lemma 3.2.6 (Joining Environments)**.** *If $\Gamma; \Delta \vdash \diamond$ and $\Gamma; \Delta' \vdash \diamond$, then $\Gamma; \Delta, \Delta' \vdash \diamond$.*

Proof. By induction on the size of Δ', using Lemma 3.2.5 (point 1). □

Notation 3.2.1 (Environment Entry η)**.** We define an environment entry η to be either a type environment entry μ or a formula F.

Notation 3.2.2 (Environment Join $\bullet$)**.** We introduce the following notation for environment join:

$$(\Gamma;\Delta)\bullet\mu \triangleq \begin{cases}\Gamma, x:\psi(T);\Delta, \mathit{forms}(x:T) & \text{if } \mu = x:T\\ \Gamma,\mu;\Delta & \text{otherwise}\end{cases}$$

$$(\Gamma;\Delta)\bullet F \triangleq \Gamma;\Delta, F$$

$$(\Gamma;\Delta)\bullet(\Gamma';\Delta') \triangleq \Gamma,\Gamma';\Delta,\Delta'$$

Lemma 3.2.7 (Weakening)**.** *If* $(\Gamma;\Delta)\bullet(\Gamma';\Delta')\vdash\mathcal{J}$ *and* $(\Gamma;\Delta)\bullet\eta\bullet(\Gamma';\Delta')\vdash\diamond$*, then* $(\Gamma;\Delta)\bullet\eta\bullet(\Gamma';\Delta')\vdash\mathcal{J}$*.*

Proof. By induction on the derivation of $(\Gamma;\Delta)\bullet(\Gamma';\Delta')\vdash\mathcal{J}$. □

Lemma 3.2.8 (Properties of Rewriting)**.** *The following statements hold true:*

1. *If* $\Gamma;\Delta\vdash\diamond$ *and* $\Delta'\subseteq\Delta$*, then* $\Gamma;\Delta\hookrightarrow\Gamma;\Delta'$*.*
2. *If* $\Gamma;\Delta_1\hookrightarrow\Gamma;\Delta_1'$ *and* $\Gamma;\Delta_2\hookrightarrow\Gamma;\Delta_2'$*, then* $\Gamma;\Delta_1,\Delta_2\hookrightarrow\Gamma;\Delta_1',\Delta_2'$*.*
3. *If* $\Gamma;\Delta\hookrightarrow\Gamma;\Delta'$ *and* $\Gamma;\Delta'\hookrightarrow\Gamma;\Delta''$*, then* $\Gamma;\Delta\hookrightarrow\Gamma;\Delta''$*.*
4. *If* $\Gamma;\Delta\hookrightarrow\Gamma;!\Delta'$*, then* $\Gamma;\Delta\hookrightarrow\Gamma;!\Delta',!\Delta'$*.*

Proof. We proceed as follows:

1. Since $\Delta'\subseteq\Delta$, we know that $\Gamma;\Delta'\vdash\diamond$ by Lemma 3.2.5. By Lemma 3.2.2 we know that $\Delta\vdash\Delta'$, hence $\Gamma;\Delta\hookrightarrow\Gamma;\Delta'$ by (Rewrite).
2. By inverting (Rewrite) we know that $\Gamma;\Delta_1\vdash\diamond$, $\Gamma;\Delta_1'\vdash\diamond$, $\Gamma;\Delta_2\vdash\diamond$, $\Gamma;\Delta_2'\vdash\diamond$, $\Delta_1\vdash\Delta_1'$ and $\Delta_2\vdash\Delta_2'$. By Lemma 3.2.6 we have $\Gamma;\Delta_1,\Delta_2\vdash\diamond$ and $\Gamma;\Delta_1',\Delta_2'\vdash\diamond$. By ($\otimes$-Right) we get $\Delta_1,\Delta_2\vdash\Delta_1',\Delta_2'$ from $\Delta_1\vdash\Delta_1'$ and $\Delta_2\vdash\Delta_2'$, hence we conclude $\Gamma;\Delta_1,\Delta_2\hookrightarrow\Gamma;\Delta_1',\Delta_2'$ by (Rewrite).
3. By inverting (Rewrite) we know that $\Gamma;\Delta\vdash\diamond$, $\Gamma;\Delta'\vdash\diamond$, $\Gamma;\Delta''\vdash\diamond$, $\Delta\vdash\Delta'$ and $\Delta'\vdash\Delta''$. By Lemma 3.2.3 we know that $\Delta\vdash\Delta'$ and $\Delta'\vdash\Delta''$ imply $\Delta\vdash\Delta''$, hence we conclude $\Gamma;\Delta\hookrightarrow\Gamma;\Delta''$ by (Rewrite).
4. By inverting (Rewrite) we know that $\Gamma;\Delta\vdash\diamond$, $\Gamma;!\Delta'\vdash\diamond$ and $\Delta\vdash!\Delta'$. Since $\Gamma;!\Delta'\vdash\diamond$ implies $\mathit{fnfv}(!\Delta')\subseteq\mathit{dom}(\Gamma)$ by Lemma 3.2.5, we get $\Gamma;!\Delta',!\Delta'\vdash\diamond$ by multiple applications of (Form Env Entry). By Lemma 3.2.4 we know that $\Delta\vdash!\Delta'$ implies $\Delta\vdash!\Delta',!\Delta'$, hence we conclude $\Gamma;\Delta\hookrightarrow\Gamma;!\Delta',!\Delta'$ by using (Rewrite).

□

Lemma 3.2.9 (Rewrite Weak)**.** *If* $\Gamma; \Delta' \vdash \mathcal{J}$ *and* $\Gamma; \Delta \hookrightarrow \Gamma; \Delta'$*, then* $\Gamma; \Delta \vdash \mathcal{J}$*.*

Proof. We distinguish on $\mathcal{J}$:

1. $\mathcal{J} = \diamond$: This case follows immediately by the definition of (Rewrite).

2. $\mathcal{J} = T$: By definition of rule (Type) we know that $\Gamma; \Delta' \vdash \diamond$ and $\mathit{fnfv}(T) \subseteq \mathit{dom}(\Gamma)$. We also know that $\Gamma; \Delta \hookrightarrow \Gamma; \Delta'$, which by (Rewrite) implies $\Gamma; \Delta \vdash \diamond$. Since $\Gamma; \Delta \vdash \diamond$ and $\mathit{fnfv}(T) \subseteq \mathit{dom}(\Gamma)$, we conclude $\Gamma; \Delta \vdash T$ by using (Type).

3. $\mathcal{J} = F$: By definition of rule (Derive) we know that $\Gamma; \Delta' \vdash \diamond$, $\mathit{fnfv}(F) \subseteq \mathit{dom}(\Gamma)$ and $\Delta' \vdash F$. We also know that $\Gamma; \Delta \hookrightarrow \Gamma; \Delta'$, which by (Rewrite) implies $\Gamma; \Delta \vdash \diamond$ and $\Delta \vdash \Delta'$. We can apply Lemma 3.2.3 to $\Delta \vdash \Delta'$ and $\Delta' \vdash F$, and get $\Delta \vdash F$. Since $\Gamma; \Delta \vdash \diamond$, $\mathit{fnfv}(F) \subseteq \mathit{dom}(\Gamma)$ and $\Delta \vdash F$, we conclude $\Gamma; \Delta \vdash F$ by (Derive).

4. $\mathcal{J} = T :: k$: We proceed by induction on the derivation of $\Gamma; \Delta' \vdash T :: k$. The cases (Kind Var) and (Kind Unit) follow immediately by proof part (1). The case (Kind Refine Public) follows by proof part (2) and an application of the induction hypothesis. All other cases contain a rewriting statement of the form $\Gamma; \Delta' \hookrightarrow \Gamma; \Delta''$ among their hypotheses. By Lemma 3.2.8 (point 3) it follows that $\Gamma; \Delta \hookrightarrow \Gamma; \Delta''$, thus allowing us to immediately conclude by applying the original rule.

5. $\mathcal{J} = T <: U$: By induction on the derivation of $\Gamma; \Delta' \vdash T <: U$, using the same reasoning as in the previous case.

6. $\mathcal{J} = E : T$: By induction on the derivation of $\Gamma; \Delta' \vdash E : T$, using the same reasoning as in the previous cases.

□

Lemma 3.2.10 (Rewriting and Variables)**.** *If* $x \notin \mathit{dom}(\Gamma)$ *and* $\Gamma; \Delta \hookrightarrow \Gamma; \Delta'$*, then* $x \notin \mathit{fv}(\Delta')$*.*

Proof. Immediate by Lemma 3.2.5 (point 1), since $\Gamma; \Delta \hookrightarrow \Gamma; \Delta'$ implies $\Gamma; \Delta' \vdash \diamond$ by inverting rule (Rewrite). □

Lemma 3.2.11 (Soundness of ψ)**.** *For every type* T*, we have* $\mathit{forms}(x : \psi(T)) = \emptyset$*.*

Proof. By induction on the structure of T. □

Lemma 3.2.12 (Idempotent ψ)**.** *For every type* T*, we have* $\psi(\psi(T)) = \psi(T)$*.*

Proof. By induction on the structure of T. □

3.2.3 Properties of kinding and subtyping

Lemma 3.2.13 (Bare Kinds). *If $\Gamma; \Delta \vdash T :: k$, then there exist $!\Delta'$ and Δ'' such that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta', \Delta''$ and $\Gamma; !\Delta' \vdash \psi(T) :: k$. Moreover, if $k = \mathsf{tnt}$, we can also require $\Delta'' \vdash \mathit{forms}(x : T)$ for any $x \notin \mathit{dom}(\Gamma)$.*

Proof. By induction on the derivation of $\Gamma; \Delta \vdash T :: k$:

Case (Kind Var): assume that $\Gamma; \Delta \vdash \alpha :: k$ by the premise $(\alpha :: k) \in \Gamma$ with $\Gamma; \Delta \vdash \diamond$. Since $\mathit{forms}(x : \alpha) = \emptyset$ and $\psi(\alpha) = \alpha$, we just need to show that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$ for some $!\Delta'$ such that $\Gamma; !\Delta' \vdash \alpha :: k$. We note that $\Gamma; \Delta \vdash \diamond$ implies $\Gamma; \emptyset \vdash \diamond$ by Lemma 3.2.5 (point 1), hence $\Gamma; \emptyset \vdash \alpha :: k$ by (Kind Var). Since $\Gamma; \Delta \hookrightarrow \Gamma; \emptyset$ by Lemma 3.2.8 (point 1), this is the desired conclusion.

Case (Kind Unit): assume that $\Gamma; \Delta \vdash \mathsf{unit} :: k$ by the premise $\Gamma; \Delta \vdash \diamond$. Since $\mathit{forms}(x : \mathsf{unit}) = \emptyset$ and $\psi(\mathsf{unit}) = \mathsf{unit}$, we just need to show that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$ for some $!\Delta'$ such that $\Gamma; !\Delta' \vdash \mathsf{unit} :: k$. We note that $\Gamma; \Delta \vdash \diamond$ implies $\Gamma; \emptyset \vdash \diamond$ by Lemma 3.2.5 (point 1), hence $\Gamma; \emptyset \vdash \mathsf{unit} :: k$ by (Kind Unit). Since $\Gamma; \Delta \hookrightarrow \Gamma; \emptyset$ by Lemma 3.2.8 (point 1), this is the desired conclusion.

Case (Kind Fun): assume that $\Gamma; \Delta \vdash x : T \to U :: k$ by the premises $\Gamma; !\Delta_1 \vdash T :: \overline{k}$ and $\Gamma, x : \psi(T); !\Delta_2 \vdash U :: k$ with $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2$. Since $\psi(x : T \to U) = x : T \to U$ and $\mathit{forms}(x : (x : T \to U)) = \emptyset$, the conclusion is immediate.

Case (Kind Refine Public): assume that $\Gamma; \Delta \vdash \{x : T \mid F\} :: \mathsf{pub}$ by the premises $\Gamma; \Delta \vdash \{x : T \mid F\}$ and $\Gamma; \Delta \vdash T :: \mathsf{pub}$. By inductive hypothesis $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, \Delta_2$ for some $!\Delta_1, \Delta_2$ such that $\Gamma; !\Delta_1 \vdash \psi(T) :: \mathsf{pub}$. Since $\psi(\{x : T \mid F\}) = \psi(T)$ by definition, we can conclude.

Case (Kind Refine Tainted): We know that $\Gamma; \Delta \vdash \{x : T \mid F\} :: \mathsf{tnt}$ by the premises $\Gamma; \Delta_1 \vdash \psi(\{x : T \mid F\}) :: \mathsf{tnt}$ and $\Gamma, x : \psi(\{x : T \mid F\}); \Delta_2 \vdash \mathit{forms}(x : \{x : T \mid F\})$ with $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$. We apply the inductive hypothesis to get $\Gamma; \Delta_1 \hookrightarrow \Gamma; !\Delta_{11}, \Delta_{12}$ for some $!\Delta_{11}$ and Δ_{12} such that $\Gamma; !\Delta_{11} \vdash \psi(\psi(\{x : T \mid F\})) :: \mathsf{tnt}$ and $\Delta_{12} \vdash \mathit{forms}(x : \psi(\{x : T \mid F\}))$. Note that the former judgement is equivalent to $\Gamma; !\Delta_{11} \vdash \psi(\{x : T \mid F\}) :: \mathsf{tnt}$ by Lemma 3.2.12. By inverting (Derive) we have $\Delta_2 \vdash \mathit{forms}(x : \{x : T \mid F\})$. Since $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_{11}, \Delta_2$ by Lemma 3.2.8, we can conclude.

The cases for rules (Kind Pair), (Kind Sum) and (Kind Rec) are identical to the case for (Kind Fun). □

Lemma 3.2.14 (Bare Kinds Reverse). *If $\Gamma; \Delta \vdash \psi(T) :: \mathsf{pub}$ and $\Gamma; \Delta \vdash T$, then $\Gamma; \Delta \vdash T :: \mathsf{pub}$.*

Proof. By induction on the structure of T. In most cases $T = \psi(T)$, allowing us to immediately conclude. In the case where $T = \{x : U \mid F\}$, assume that

$\Gamma;\Delta \vdash \psi(\{x : U \mid F\}) :: \mathsf{pub}$ and $\Gamma;\Delta \vdash \{x : U \mid F\}$. We observe that the latter implies $\Gamma;\Delta \vdash U$. By definition we have $\psi(\{x : U \mid F\}) = \psi(U)$, hence by inductive hypothesis we get $\Gamma;\Delta \vdash U :: \mathsf{pub}$. Since $\Gamma;\Delta \vdash \{x : U \mid F\}$ by hypothesis and $\Gamma;\Delta \vdash U :: \mathsf{pub}$, we conclude $\Gamma;\Delta \vdash \{x : U \mid F\} :: \mathsf{pub}$ by (Kind Refine Public). □

Lemma 3.2.15 (Bare Subtypes)**.** *If $\Gamma;\Delta \vdash T <: U$, then there exist $!\Delta'$ and Δ'' such that $\Gamma;\Delta \hookrightarrow \Gamma;!\Delta',\Delta''$ and $\Gamma;!\Delta' \vdash \psi(T) <: \psi(U)$ and $\Delta'', \mathit{forms}(x : T) \vdash \mathit{forms}(x : U)$ for any $x \notin \mathit{dom}(\Gamma)$.*

Proof. By induction on the derivation of $\Gamma;\Delta \vdash T <: U$:

Case (Sub Refl): assume that $\Gamma;\Delta \vdash T <: T$ by the premise $\Gamma;\Delta \vdash T$. Since $\Gamma;\emptyset \vdash \psi(T)$ by Lemma 3.2.5 (point 3), we have $\Gamma;\emptyset \vdash \psi(T) <: \psi(T)$ by (Sub Refl). Moreover, we note that $\mathit{forms}(x : T) \vdash \mathit{forms}(x : T)$. This leads to the desired conclusion, since $\Gamma;\Delta \hookrightarrow \Gamma;\emptyset$ by Lemma 3.2.8 (point 1).

Case (Sub Pub Tnt): assume that $\Gamma;\Delta \vdash T <: U$ by the premises $\Gamma;\Delta_1 \vdash T :: \mathsf{pub}$ and $\Gamma;\Delta_2 \vdash U :: \mathsf{tnt}$ with $\Gamma;\Delta \hookrightarrow \Gamma;\Delta_1,\Delta_2$. We apply Lemma 3.2.13 to $\Gamma;\Delta_1 \vdash T :: \mathsf{pub}$ and we get $\Gamma;\Delta_1 \hookrightarrow \Gamma;!\Delta_{11},\Delta_{12}$ for some $!\Delta_{11},\Delta_{12}$ such that $\Gamma;!\Delta_{11} \vdash \psi(T) :: \mathsf{pub}$. Then we apply Lemma 3.2.13 to $\Gamma;\Delta_2 \vdash U :: \mathsf{tnt}$ and we get $\Gamma;\Delta_2 \hookrightarrow \Gamma;!\Delta_{21},\Delta_{22}$ for some $!\Delta_{21},\Delta_{22}$ such that $\Gamma;!\Delta_{21} \vdash \psi(U) :: \mathsf{tnt}$ and $\Gamma;\Delta_{22} \vdash \mathit{forms}(x : U)$. By an application of (Sub Pub Tnt) we then get $\Gamma;!\Delta_{11},!\Delta_{21} \vdash \psi(T) <: \psi(U)$. Now we notice that $\Gamma;\Delta \hookrightarrow \Gamma;!\Delta_{11},\Delta_{12},!\Delta_{21},\Delta_{22}$ by Lemma 3.2.8, which implies $\Gamma;\Delta \hookrightarrow \Gamma;(!\Delta_{11},!\Delta_{21}),\Delta_{22}$ again by Lemma 3.2.8, hence we conclude.

Case (Sub Fun): assume that $\Gamma;\Delta \vdash y : U_1 \to U_2 <: y : U_3 \to U_4$ by the premises $\Gamma;!\Delta_1 \vdash U_3 <: U_1$ and $\Gamma, y : \psi(U_3);!\Delta_2 \vdash U_2 <: U_4$ with $\Gamma;\Delta \hookrightarrow \Gamma;!\Delta_1,!\Delta_2$. Since $\psi(y : U_1 \to U_2) = y : U_1 \to U_2$ and $\psi(y : U_3 \to U_4) = y : U_3 \to U_4$ and $\mathit{forms}(x : (y : U_1 \to U_2)) = \mathit{forms}(x : (y : U_3 \to U_4)) = \emptyset$, the conclusion is immediate.

Case (Sub Refine): assume that $\Gamma;\Delta \vdash T <: U$ by the premises $\Gamma;\Delta_1 \vdash \psi(T) <: \psi(U)$ and $\Gamma, y : \psi(T);\Delta_2, \mathit{forms}(y : T) \vdash \mathit{forms}(y : U)$ with $\Gamma;\Delta \hookrightarrow \Gamma;\Delta_1,\Delta_2$. We apply the inductive hypothesis to $\Gamma;\Delta_1 \vdash \psi(T) <: \psi(U)$ and we get that there exist $!\Delta_{11},\Delta_{12}$ such that $\Gamma;!\Delta_{11} \vdash \psi(\psi(T)) <: \psi(\psi(U))$ and $\Gamma;\Delta_1 \hookrightarrow \Gamma;!\Delta_{11},\Delta_{12}$. The former judgement is equivalent to $\Gamma;!\Delta_{11} \vdash \psi(T) <: \psi(U)$ by Lemma 3.2.12, while by inverting rule (Derive) we have $\Delta_2, \mathit{forms}(y : T) \vdash \mathit{forms}(y : U)$; hence, to conclude we just note that $\Gamma;\Delta \hookrightarrow \Gamma;!\Delta_{11},\Delta_2$ by Lemma 3.2.8.

The cases for rules (Sub Pair), (Sub Sum) and (Sub Pos Rec) are identical to the case for (Sub Fun). □

Lemma 3.2.16 (Replacing Unrefined Bindings)**.** *For all $\mathcal{J} \in \{\diamond, U, F, U :: k, U <: U'\}$ it holds that if $\Gamma, x : \psi(T), \Gamma';\Delta \vdash \mathcal{J}$ and $\Gamma;\emptyset \vdash \psi(T')$, then $\Gamma, x : \psi(T'), \Gamma';\Delta \vdash \mathcal{J}$. Moreover, the depth of the two derivations is the same.*

Proof. We prove all statements separately by induction on the derivation of $\Gamma, x : \psi(T), \Gamma'; \Delta \vdash \mathcal{J}$, making use of Lemma 3.2.5 when needed. $\square$

Definition 3.2.3 (Compartmental Notation for Environments)**.** Let $\Gamma[(\mu_i)^{i\in\{1,\dots,n\}}]$ denote the environment obtained by inserting the entries $\mu_1, \dots, \mu_n$ at fixed positions between the entries of the environment Γ.

Lemma 3.2.17 (Type Variables and Kinding)**.** *For all* $\Gamma = \Gamma_0[(\alpha_i)^{i\in\{1,\dots,n\}}]$ *and* $\hat{\Gamma} = \Gamma_0[(\alpha_i :: k_i)^{i\in\{1,\dots,n\}}]$ *it holds that:*

1. *dom*$(\Gamma) =$ *dom*$(\hat{\Gamma})$*;*
2. $\Gamma; \Delta \vdash \diamond$ *if and only if* $\hat{\Gamma}; \Delta \vdash \diamond$*;*
3. $\Gamma; \Delta \hookrightarrow \Gamma; \Delta'$ *if and only if* $\hat{\Gamma}; \Delta \hookrightarrow \hat{\Gamma}; \Delta'$*;*
4. $\Gamma; \Delta \vdash T$ *if and only if* $\hat{\Gamma}; \Delta \vdash T$*;*
5. $\Gamma; \Delta \vdash F$ *if and only if* $\hat{\Gamma}; \Delta \vdash F$*;*
6. *If* $\Gamma; \Delta \vdash T :: k$*, then* $\hat{\Gamma}; \Delta \vdash T :: k$*.*

Proof. We proceed as follows:

1. We note that $\mathit{dom}(\alpha_i) = \mathit{dom}(\alpha_i :: k_i)$ by the definition of *dom* and we easily conclude.
2. $\Gamma; \Delta$ and $\hat{\Gamma}; \Delta$ only differ in α_i and $\alpha_i :: k_i$ respectively. The statement follows noting that $\mathit{dom}(\alpha_i) = \{\alpha_i\} = \mathit{dom}(\alpha_i :: k_i)$.
3. By definition of (Rewrite), using (2).
4. By definition of (Type), using (1) and (2).
5. By definition of (Derive), using (1) and (2).
6. By induction on the derivation of $\Gamma; \Delta \vdash T :: k$, using the previous statements.

$\square$

Lemma 3.2.18 (Public Down/Tainted Up)**.** *For all environments* $\Gamma; \Delta$ *and types* T, T' *it holds that:*

1. *If* $\Gamma; \Delta \vdash T <: T'$ *and* $\Gamma; \Delta' \vdash T' :: \mathsf{pub}$*, then* $\Gamma; \Delta, \Delta' \vdash T :: \mathsf{pub}$*.*
2. *If* $\Gamma; \Delta \vdash T <: T'$ *and* $\Gamma; \Delta' \vdash T :: \mathsf{tnt}$*, then* $\Gamma; \Delta, \Delta' \vdash T' :: \mathsf{tnt}$*.*

Proof. The lemma is an instance (for $n = 0$) of the following more general statement: For all environments $\Gamma; \Delta$ and types T, T' such that $\Gamma = \Gamma_0[(\alpha_i)^{i\in\{1,\dots,n\}}]$ and $\hat{\Gamma} = \Gamma_0[(\alpha_i :: k_i)^{i\in\{1,\dots,n\}}]$ it holds:

1. If $\Gamma; \Delta \vdash T <: T'$ and $\hat{\Gamma}; \Delta' \vdash T' :: \mathsf{pub}$, then $\hat{\Gamma}; \Delta, \Delta' \vdash T :: \mathsf{pub}$.
2. If $\Gamma; \Delta \vdash T <: T'$ and $\hat{\Gamma}; \Delta' \vdash T :: \mathsf{tnt}$, then $\hat{\Gamma}; \Delta, \Delta' \vdash T' :: \mathsf{tnt}$.

Both statements are proved by simultaneous induction on the derivation of $\Gamma; \Delta \vdash T <: T'$. We distinguish the last applied subtyping rule and we often implicitly appeal to Lemma 3.2.5 and Lemma 3.2.17. Notice in particular that, by using Lemma 3.2.5 and Lemma 3.2.17, we can derive both $\Gamma; \Delta, \Delta' \vdash \diamond$ and $\hat{\Gamma}; \Delta, \Delta' \vdash \diamond$.

Case (SUB REFL): In this case $T = T'$, hence we know in the two cases that:

1. $\hat{\Gamma}; \Delta' \vdash T :: \mathsf{pub}$. As seen above, we know that $\hat{\Gamma}; \Delta, \Delta' \vdash \diamond$, hence $\hat{\Gamma}; \Delta, \Delta' \vdash T :: \mathsf{pub}$ follows by Lemma 3.2.7.
2. $\hat{\Gamma}; \Delta' \vdash T' :: \mathsf{tnt}$. Using the same reasoning as in the previous case we can conclude that $\hat{\Gamma}; \Delta, \Delta' \vdash T' :: \mathsf{tnt}$ follows by Lemma 3.2.7.

Case (SUB PUB TNT): In this case it holds that $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$ such that $\Gamma; \Delta_1 \vdash T :: \mathsf{pub}$ and $\Gamma; \Delta_2 \vdash T' :: \mathsf{tnt}$. Notice again that $\Gamma; \Delta, \Delta' \vdash \diamond$ as before.

In the proof of statement (1) we need to show that $\hat{\Gamma}; \Delta, \Delta' \vdash T :: \mathsf{pub}$. By Lemma 3.2.8 we know that $\Gamma; \Delta, \Delta' \hookrightarrow \Gamma; \Delta_1$. We derive that $\Gamma; \Delta, \Delta' \vdash T :: \mathsf{pub}$ by an application of Lemma 3.2.9. We apply Lemma 3.2.17 to conclude that $\hat{\Gamma}; \Delta, \Delta' \vdash T :: \mathsf{pub}$.

In the proof of statement (2) we need to show that $\Gamma; \Delta, \Delta' \vdash T' :: \mathsf{tnt}$. By Lemma 3.2.8 we know that $\Gamma; \Delta, \Delta' \hookrightarrow \Gamma; \Delta_2$. We conclude by an application of Lemma 3.2.9 that $\Gamma; \Delta, \Delta' \vdash T' :: \mathsf{tnt}$. Using Lemma 3.2.17 we can conclude that $\hat{\Gamma}; \Delta, \Delta' \vdash T' :: \mathsf{tnt}$.

Case (SUB REFINE): In this case we know that $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$ such that $\Gamma; \Delta_1 \vdash \psi(T) <: \psi(T')$ and $\Gamma, y : \psi(T); \Delta_2, \mathit{forms}(y : T) \vdash \mathit{forms}(y : T')$.

We show both statements separately. We first note that by Lemma 3.2.5 we know that $\Gamma; \emptyset \vdash T$ and $\Gamma; \emptyset \vdash T'$ and thus by Lemma 3.2.17 $\hat{\Gamma}; \emptyset \vdash T$ and $\hat{\Gamma}; \emptyset \vdash T'$.

1. By Lemma 3.2.13 we know that there exist Δ_1', Δ_2' such that:
   - $\hat{\Gamma}; \Delta' \hookrightarrow \hat{\Gamma}; !\Delta_1', \Delta_2'$,
   - $\hat{\Gamma}; !\Delta_1' \vdash \psi(T') :: \mathsf{pub}$.

   We can apply the induction hypothesis to derive that:

   $$\hat{\Gamma}; \Delta_1, !\Delta_1' \vdash \psi(T) :: \mathsf{pub}.$$

   By Lemma 3.2.14 we can immediately derive that:

   $$\hat{\Gamma}; \Delta_1, !\Delta_1' \vdash T :: \mathsf{pub}.$$

   We can derive that $\hat{\Gamma}; \Delta, \Delta' \hookrightarrow \hat{\Gamma}; \Delta_1, !\Delta_1'$ using Lemma 3.2.8 in combination with Lemma 3.2.17, hence we conclude $\hat{\Gamma}; \Delta, \Delta' \vdash T :: \mathsf{pub}$ by Lemma 3.2.9.

2. By Lemma 3.2.13 we know that there exist Δ_1', Δ_2' such that:
   - $\hat{\Gamma}; \Delta' \hookrightarrow \hat{\Gamma}; !\Delta_1', \Delta_2'$,
   - $\hat{\Gamma}; !\Delta_1' \vdash \psi(T) :: \mathsf{tnt}$, and
   - $\Delta_2' \vdash \mathit{forms}(y : T)$ for some $y \notin \mathit{dom}(\Gamma)$.

   We can apply the induction hypothesis to derive that:

   $$\hat{\Gamma}; \Delta_1, !\Delta_1' \vdash \psi(T') :: \mathsf{tnt}.$$

   If $\psi(T') = T'$, we observe that $\hat{\Gamma}; \Delta, \Delta' \hookrightarrow \hat{\Gamma}; \Delta_1, !\Delta_1'$ by Lemma 3.2.8 in combination with Lemma 3.2.17, hence we conclude $\hat{\Gamma}; \Delta, \Delta' \vdash T :: \mathsf{tnt}$ by Lemma 3.2.9.

   Otherwise, we know that T' is refined. We stated that $\Gamma, y : \psi(T); \Delta_2, \mathit{forms}(y : T) \vdash \mathit{forms}(y : T')$, thus, by inverting (Derive), we know that $\Delta_2, \mathit{forms}(y : T) \vdash \mathit{forms}(y : T')$. Using Lemma 3.2.3 we get:

   $$\Delta_2', \Delta_2 \vdash \mathit{forms}(y : T'),$$

   hence, by applying (Derive) and some simple observations, we know that $\hat{\Gamma}, y : \psi(T'); \Delta_2', \Delta_2 \vdash \mathit{forms}(y : T')$. By (Kind Refine Tainted) we then get:

   $$\frac{\hat{\Gamma}; \Delta_1, !\Delta_1' \vdash \psi(T') :: \mathsf{tnt} \qquad \hat{\Gamma}, y : \psi(T'); \Delta_2', \Delta_2 \vdash \mathit{forms}(y : T')}{\hat{\Gamma}; !\Delta_1', \Delta_2', \Delta_2 \vdash T' :: \mathsf{tnt}}$$

   By Lemma 3.2.8 in combination with Lemma 3.2.17, we know that $\hat{\Gamma}; \Delta, \Delta' \hookrightarrow \hat{\Gamma}; \Delta_1, \Delta_2, !\Delta_1', \Delta_2'$, hence we conclude $\hat{\Gamma}; \Delta, \Delta' \vdash T' :: \mathsf{tnt}$ by Lemma 3.2.9.

Case (Sub Sum): In this case we know that $T = T_1 + T_2$ and $T' = T_1' + T_2'$ and $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2$ such that $\Gamma; !\Delta_i \vdash T_i <: T_i'$ for $i \in \{1, 2\}$.

1. By the definition of the only applicable kinding rule (Kind Sum) we also know that $\hat{\Gamma}; \Delta' \hookrightarrow \hat{\Gamma}; !\Delta_1', !\Delta_2'$ such that $\hat{\Gamma}; !\Delta_i' \vdash T_i' :: \mathsf{pub}$ for $i \in \{1, 2\}$. We apply the induction hypothesis twice and derive that $\hat{\Gamma}; !\Delta_i, !\Delta_i' \vdash T_i :: \mathsf{pub}$. Since we know that $\hat{\Gamma}; \Delta, \Delta' \hookrightarrow \hat{\Gamma}; !\Delta_1, !\Delta_2, !\Delta_1', !\Delta_2' = \hat{\Gamma}; !\Delta_1, !\Delta_1', !\Delta_2, !\Delta_2'$ by Lemma 3.2.8 and Lemma 3.2.17, we conclude $\hat{\Gamma}; \Delta, \Delta' \vdash T :: \mathsf{pub}$ by an application of (Kind Sum).
2. Analogous to the case for statement (1).

Case (Sub Pos Rec): We know that $T = \mu\alpha. U$ and $T' = \mu\alpha. U'$ and $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1$ such that $\Gamma, \alpha; !\Delta_1 \vdash U <: U'$ and α occurs only positively in U and U'.

1. By the definition of the only applicable kinding rule (Kind Rec) we also know that $\hat{\Gamma}; \Delta' \hookrightarrow \hat{\Gamma}; !\Delta_1'$ such that $\hat{\Gamma}, \alpha :: \mathsf{pub}; !\Delta_1' \vdash U' :: \mathsf{pub}$. We define $\alpha_{n+1} \triangleq \alpha$

and $\Gamma' \triangleq \Gamma, \alpha = \Gamma_0[(\alpha_i)^{i \in \{1,\dots,n+1\}}]$. Furthermore, we define $k_{n+1} \triangleq \mathsf{pub}$ and $\hat{\Gamma}' \triangleq \hat{\Gamma}, \alpha :: \mathsf{pub} = \Gamma_0[(\alpha_i :: k_i)^{i \in \{1,\dots,n+1\}}]$. We can thus apply the induction hypothesis and derive that $\hat{\Gamma}'; !\Delta_1, !\Delta'_1 \vdash U :: \mathsf{pub}$, which is equivalent to $\hat{\Gamma}, \alpha :: \mathsf{pub}; !\Delta_1, !\Delta'_1 \vdash U :: \mathsf{pub}$. Since we know that $\hat{\Gamma}; \Delta, \Delta' \hookrightarrow \hat{\Gamma}; !\Delta_1, !\Delta'_1$ by Lemma 3.2.8 and Lemma 3.2.17, we conclude $\hat{\Gamma}; \Delta, \Delta' \vdash \mu\alpha.U :: \mathsf{pub}$ by an application of (Kind Rec).

2. Analogous to the case for statement (1).

Case (Sub Pair): In this case $T = x : T_1 * T_2$ and $T' = x : T'_1 * T'_2$. We know that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2$ such that $\Gamma; !\Delta_1 \vdash T_1 <: T'_1$ and $\Gamma, x : \psi(T_1); !\Delta_2 \vdash T_2 <: T'_2$.

1. By the only applicable kinding rule (Kind Pair), we have $\hat{\Gamma}; \Delta' \hookrightarrow \hat{\Gamma}; !\Delta'_1, !\Delta'_2$ such that $\hat{\Gamma}; !\Delta'_1 \vdash T'_1 :: \mathsf{pub}$ and $\hat{\Gamma}, x : \psi(T'_1); !\Delta'_2 \vdash T'_2 :: \mathsf{pub}$.
   We apply the induction hypothesis to derive that:
   $$\hat{\Gamma}; !\Delta_1, !\Delta'_1 \vdash T_1 :: \mathsf{pub}.$$
   We apply Lemma 3.2.16 to transform $\hat{\Gamma}, x : \psi(T'_1); !\Delta'_2 \vdash T'_2 :: \mathsf{pub}$ into:
   $$\hat{\Gamma}, x : \psi(T_1); !\Delta'_2 \vdash T'_2 :: \mathsf{pub},$$
   allowing us to apply the induction hypothesis a second time to derive that:
   $$\hat{\Gamma}, x : \psi(T_1); !\Delta_2, !\Delta'_2 \vdash T_2 :: \mathsf{pub}.$$
   We conclude $\hat{\Gamma}; \Delta, \Delta' \vdash T :: \mathsf{pub}$ by an application of (Kind Pair), since we know that $\hat{\Gamma}; \Delta, \Delta' \hookrightarrow \hat{\Gamma}; !\Delta_1, !\Delta'_1, !\Delta_2, !\Delta'_2$ by Lemma 3.2.8 and Lemma 3.2.17.
2. Analogous to the case for statement (1).

Case (Sub Fun): In this case $T = x : T_1 \to T_2$ and $T' = x : T'_1 \to T'_2$. We know that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2$ such that $\Gamma; !\Delta_1 \vdash T'_1 <: T_1$ and $\Gamma, x : \psi(T'_1); !\Delta_2 \vdash T_2 <: T'_2$.

1. By the only applicable kinding rule (Kind Fun), we have $\hat{\Gamma}; \Delta' \hookrightarrow \hat{\Gamma}; !\Delta'_1, !\Delta'_2$ such that $\hat{\Gamma}; !\Delta'_1 \vdash T'_1 :: \mathsf{tnt}$ and $\hat{\Gamma}, x : \psi(T'_1); !\Delta'_2 \vdash T'_2 :: \mathsf{pub}$.
   We apply the induction hypothesis (2) to derive that:
   $$\hat{\Gamma}; !\Delta_1, !\Delta'_1 \vdash T_1 :: \mathsf{tnt}.$$
   We apply the induction hypothesis (1) to derive that:
   $$\hat{\Gamma}, x : \psi(T'_1); !\Delta_2, !\Delta'_2 \vdash T_2 :: \mathsf{pub}.$$
   We apply Lemma 3.2.16 to transform $\hat{\Gamma}, x : \psi(T'_1); !\Delta_2, !\Delta'_2 \vdash T_2 :: \mathsf{pub}$ into:
   $$\hat{\Gamma}, x : \psi(T_1); !\Delta_2, !\Delta'_2 \vdash T_2 :: \mathsf{pub}.$$
   We conclude $\hat{\Gamma}; \Delta, \Delta' \vdash T :: \mathsf{pub}$ by an application of (Kind Fun), using the fact that $\hat{\Gamma}; \Delta, \Delta' \hookrightarrow \hat{\Gamma}; !\Delta_1, !\Delta'_1, !\Delta_2, !\Delta'_2$ by Lemma 3.2.8 and Lemma 3.2.17.

2. Analogous to the case for statement (1).

□

Lemma 3.2.19 (Public Tainted). *For all environments* $\Gamma;\Delta$ *and types* T *we have:*

1. $\Gamma;\Delta \vdash T :: \mathsf{pub}$ *if and only if* $\Gamma;\Delta \vdash T <: \mathsf{Un}$.
2. $\Gamma;\Delta \vdash T :: \mathsf{tnt}$ *if and only if* $\Gamma;\Delta \vdash \mathsf{Un} <: T$.

Proof. By definition $\mathsf{Un} \triangleq \mathsf{unit}$ and thus by (Kind Unit) it holds that $\Gamma;\emptyset \vdash \mathsf{Un} :: \mathsf{pub}$ and $\Gamma;\emptyset \vdash \mathsf{Un} :: \mathsf{tnt}$. We can immediately prove the forward implication by applying the subtyping rule (Sub Pub Tnt), since $\Gamma;\Delta \hookrightarrow \Gamma;\Delta$ by Lemma 3.2.8. The reverse implication follows immediately by Lemma 3.2.18. □

Lemma 3.2.20 (Subtyping and ψ). *The following statements hold true:*

1. *If* $\Gamma;\emptyset \vdash T$, *then* $\Gamma;\emptyset \vdash T <: \psi(T)$.
2. *If* $\Gamma;\Delta \vdash \psi(T) <: U$ *and* $\Gamma;\emptyset \vdash T$, *then* $\Gamma;\Delta \vdash T <: U$.

Proof. We proceed as follows:

1. By induction on the structure of T:
   - Whenever $T = \psi(T)$, we can conclude by an application of (Sub Refl).
   - Otherwise, we know that $T = \{x : U \mid F\}$. We know that $\Gamma;\emptyset \vdash T$, hence $\Gamma;\emptyset \vdash \psi(T)$ by Lemma 3.2.5 (point 3). Applying (Sub Refl) lets us derive that $\Gamma;\emptyset \vdash \psi(T) <: \psi(T)$, which is equivalent to $\Gamma;\emptyset \vdash \psi(T) <: \psi(\psi(T))$ by Lemma 3.2.12. Furthermore, we know that $\mathit{forms}(y : \psi(T)) = \emptyset$ by Lemma 3.2.11, hence we have $\Gamma, x : \psi(T); \mathit{forms}(y : T) \vdash \mathit{forms}(y : \psi(T))$. We thus conclude $\Gamma;\emptyset \vdash T <: \psi(T)$ by an application of (Sub Refine).

2. By induction on the derivation of $\Gamma;\Delta \vdash \psi(T) <: U$. We distinguish upon the last applied subtyping rule:
   - In the case where the last applied rule was (Sub Fun), (Sub Pair), (Sub Sum), or (Sub Pos Rec) we know that $T = \psi(T)$ and we can immediately conclude.
   - (Sub Refl): In this case we know that $U = \psi(T)$. We can thus conclude by an application of statement (1) and Lemma 3.2.7.
   - (Sub Pub Tnt): In this case we know that there exist Δ_1, Δ_2 such that $\Gamma;\Delta \hookrightarrow \Gamma;\Delta_1,\Delta_2$ and $\Gamma;\Delta_1 \vdash \psi(T) :: \mathsf{pub}$ and $\Gamma;\Delta_2 \vdash U :: \mathsf{tnt}$. By Lemma 3.2.14 we thus know that $\Gamma;\Delta_1 \vdash T :: \mathsf{pub}$, allowing us to conclude by an application of (Sub Pub Tnt).

- (Sub Refine): In this case we know that U must be refined. Furthermore, we know that there must exist Δ_1, Δ_2 such that $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$ and $\Gamma; \Delta_1 \vdash \psi(\psi(T)) <: \psi(U)$ and $\Gamma, x : \psi(\psi(T)); \Delta_2, \mathit{forms}(x : \psi(T)) \vdash \mathit{forms}(x : U)$. Note that by Lemma 3.2.12 we know that $\psi(\psi(T)) = \psi(T)$ and by Lemma 3.2.11 we know that $\mathit{forms}(x : \psi(T)) = \emptyset$. Thus, it follows that $\Gamma; \Delta_1 \vdash \psi(T) <: \psi(U)$ and $\Gamma, x : \psi(T); \Delta_2 \vdash \mathit{forms}(x : U)$. We can apply Lemma 3.2.7 to derive that $\Gamma, x : \psi(T); \Delta_2, \mathit{forms}(x : T) \vdash \mathit{forms}(x : U)$. This allows us to conclude by an application of (Sub Refine).

□

Lemma 3.2.21 (Transitivity). *If* $\Gamma; \Delta \vdash T <: T'$ *and* $\Gamma; \Delta' \vdash T' <: T''$*, then* $\Gamma; \Delta, \Delta' \vdash T <: T''$.

Proof. By induction on the sum of the depth of the derivations of the antecedent judgements. We proceed by case analysis on the last subtyping rule R_1 applied in the derivation $\Gamma; \Delta \vdash T <: T'$ and the last applied rule R_2 in the derivation of $\Gamma; \Delta' \vdash T' <: T''$. We first note that by Lemma 3.2.5 it must be the case that $\Gamma; \Delta \vdash \diamond$ and $\Gamma; \Delta' \vdash \diamond$ and thus by Lemma 3.2.6 it holds that $\Gamma; \Delta, \Delta' \vdash \diamond$.

Case $R_1 =$ (Sub Refl): Since in this case $T = T'$, we can immediately conclude by applying Lemma 3.2.7 to $\Gamma; \Delta' \vdash T' <: T''$.

Case $R_2 =$ (Sub Refl): Since in this case $T' = T''$, we can immediately conclude by applying Lemma 3.2.7 to $\Gamma; \Delta \vdash T <: T'$.

Case $R_1 =$ (Sub Pub Tnt): By definition of (Sub Pub Tnt) it follows that $\Gamma; \Delta_1 \vdash T :: \mathsf{pub}$, $\Gamma; \Delta_2 \vdash T' :: \mathsf{tnt}$, where $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$. We can apply Lemma 3.2.18 to derive that $\Gamma; \Delta', \Delta_2 \vdash T'' :: \mathsf{tnt}$ and since we know that $\Gamma; \Delta, \Delta' \vdash \diamond$ and $\Gamma; \Delta, \Delta' \hookrightarrow \Gamma; \Delta_1, \Delta_2, \Delta'$ by Lemma 3.2.8 we apply rule (Sub Pub Tnt) to conclude.

Case $R_2 =$ (Sub Pub Tnt): By definition of (Sub Pub Tnt) it follows that $\Gamma; \Delta'_1 \vdash T' :: \mathsf{pub}$, $\Gamma; \Delta'_2 \vdash T'' :: \mathsf{tnt}$, where $\Gamma; \Delta' \hookrightarrow \Gamma; \Delta'_1, \Delta'_2$. We can apply Lemma 3.2.18 to derive that $\Gamma; \Delta, \Delta'_1 \vdash T :: \mathsf{pub}$ and since we know that $\Gamma; \Delta, \Delta' \vdash \diamond$ and $\Gamma; \Delta, \Delta' \hookrightarrow \Gamma; \Delta, \Delta'_1, \Delta'_2$ we apply rule (Sub Pub Tnt) to conclude.

Case $R_1 = R_2 =$ (Sub Sum): Follows immediately by applying the induction hypothesis twice to the premises of the applied rule (Sub Sum) and then applying (Sub Sum) to the resulting statements.

Case $R_1 = R_2 =$ (Sub Pos Rec): Follows immediately by applying the induction hypothesis to the premise of the applied rule (Sub Pos Rec) and then applying (Sub Pos Rec) to the resulting statement.

Case $R_1 =$ (Sub Refine): In this case we know that $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$ such that $\Gamma; \Delta_1 \vdash \psi(T) <: \psi(T')$ and $(\Gamma; \Delta_2) \bullet y : T \vdash \mathit{forms}(y : T')$.

We distinguish all possible rules R_2, that are not captured by previous cases:

- R_2 is either (Sub Fun), (Sub Pair), (Sub Sum), or (Sub Pos Rec): In this case we know that $\psi(T') = T'$ and we can immediately apply the induction hypothesis to derive that:

$$\Gamma; \Delta_1, \Delta' \vdash \psi(T) <: T''.$$

  By Lemma 3.2.20 it follows that:

$$\Gamma; \Delta_1, \Delta' \vdash T <: T''.$$

  By Lemma 3.2.8 we know that $\Gamma; \Delta, \Delta' \hookrightarrow \Gamma; \Delta_1, \Delta'$, allowing us to conclude by an application of Lemma 3.2.9.

- $R_2 =$ (Sub Refine): In this case we know that $\Gamma; \Delta' \hookrightarrow \Gamma; \Delta'_1, \Delta'_2$ such that $\Gamma; \Delta'_1 \vdash \psi(T') <: \psi(T'')$ and $(\Gamma; \Delta'_2) \bullet y : T' \vdash \mathit{forms}(y : T'')$.
  We can apply the induction hypothesis to $\Gamma; \Delta_1 \vdash \psi(T) <: \psi(T')$ and $\Gamma; \Delta'_1 \vdash \psi(T') <: \psi(T'')$, leading to:

$$\Gamma; \Delta_1, \Delta'_1 \vdash \psi(T) <: \psi(T'').$$

  By the definition of "$\bullet$", inverting rule (Derive), we know that $\Delta_2, \mathit{forms}(y : T) \vdash \mathit{forms}(y : T')$ and $\Delta'_2, \mathit{forms}(y : T') \vdash \mathit{forms}(y : T'')$. Using Lemma 3.2.3 we can derive that $\Delta_2, \Delta'_2, \mathit{forms}(y : T) \vdash \mathit{forms}(y : T'')$. By applying rule (Derive) and Lemma 3.2.16, we can then get:

$$(\Gamma; \Delta_2, \Delta'_2) \bullet y : T \vdash \mathit{forms}(y : T'').$$

  We also know by definition of (Sub Refine) that T and/or T' refined and T' and/or T'' refined. This implies that either T' is the only refined type or at least one type in $\{T, T''\}$ is refined. In the latter case we can immediately conclude by an application of (Sub Refine). In the former case we know that $\psi(T) = T$ and $\psi(T'') = T''$. Since $\Gamma; \Delta, \Delta' \hookrightarrow \Gamma; \Delta_1, \Delta'_1$ by Lemma 3.2.8 and $\Gamma; \Delta_1, \Delta'_1 \vdash \psi(T) <: \psi(T'') = T <: T''$, we conclude $\Gamma; \Delta, \Delta' \vdash T <: T''$ by Lemma 3.2.9.

Case $R_2 =$ (Sub Refine): In this case we know that $\Gamma; \Delta' \hookrightarrow \Gamma; \Delta'_1, \Delta'_2$ such that $\Gamma; \Delta'_1 \vdash \psi(T') <: \psi(T'')$ and $(\Gamma; \Delta'_2) \bullet y : T' \vdash \mathit{forms}(y : T'')$.

Note that all possible rules R_1 that are not captured by previous cases (Sub Fun), (Sub Pair), (Sub Sum), or (Sub Pos Rec) entail that $T = \psi(T)$ and $T' = \psi(T')$, and T'' must be refined by definition of (Sub Refine).

In particular, this means that we can apply the induction hypothesis to $\Gamma; \Delta \vdash T <: T'$ and $\Gamma; \Delta_1' \vdash \psi(T') <: \psi(T'')$, yielding:

$$\Gamma; \Delta, \Delta_1' \vdash \psi(T) <: \psi(T'').$$

By inverting (Derive) we have $\Delta_2', \mathit{forms}(y : T') \vdash \mathit{forms}(y : T'')$. By Lemma 3.2.11 we know that $\mathit{forms}(y : T') = \mathit{forms}(y : T) = \emptyset$, hence $\Delta_2', \mathit{forms}(y : T) \vdash \mathit{forms}(y : T'')$. By applying rule (Derive) and Lemma 3.2.16, we can then get:

$$\Gamma, y : \psi(T); \Delta_2', \mathit{forms}(y : T) \vdash \mathit{forms}(y : T'').$$

We conclude by an application of (Sub Refine).

Case $R_1 = R_2 =$ (Sub Fun): In this case $T = x : U \to V$, $T' = x : U' \to V'$, and $T'' = x : U'' \to V''$.

Furthermore, there must exist $\Delta_1, \Delta_2, \Delta_1', \Delta_2'$ such that:

- $\Gamma; \Delta \vdash \Gamma; !\Delta_1, !\Delta_2$,
- $\Gamma; !\Delta_1 \vdash U' <: U$,
- $\Gamma, x : \psi(U'); !\Delta_2 \vdash V <: V'$,
- $\Gamma; \Delta' \vdash \Gamma; !\Delta_1', !\Delta_2'$,
- $\Gamma; !\Delta_1' \vdash U'' <: U'$, and
- $\Gamma, x : \psi(U''); !\Delta_2' \vdash V' <: V''$.

We note that by applying Lemma 3.2.16 to the third statement we get $\Gamma, x : \psi(U''); !\Delta_2 \vdash V <: V'$, where the depth of the derivation has not changed. We apply the inductive hypothesis to $\Gamma; !\Delta_1' \vdash U'' <: U'$ and $\Gamma; !\Delta_1 \vdash U' <: U$, and we get:

$$\Gamma; !\Delta_1', !\Delta_1 \vdash U'' <: U.$$

We apply the inductive hypothesis to $\Gamma, x : \psi(U''); !\Delta_2 \vdash V <: V'$ and $\Gamma, x : \psi(U''); !\Delta_2' \vdash V' <: V''$, and we get:

$$\Gamma, x : \psi(U''); !\Delta_2, !\Delta_2' \vdash V <: V''.$$

The conclusions $\Gamma; \Delta \vdash T <: T''$ follows by (Sub Fun).

Case $R_1 = R_2 =$ (Sub Pair): Completely analogous to the previous case.

No other combination of rules is possible. □

3.2.4 Properties of substitution

Lemma 3.2.22 (Bare Types). *Let $fv(M) = \emptyset$. If $\Gamma; \Delta \vdash M : T$, then there exist $!\Delta'$ and Δ'' such that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta', \Delta''$ and $\Gamma; !\Delta' \vdash M : \psi(T)$ and $\Delta'' \vdash forms(x : T)\{M/x\}$ for any $x \notin dom(\Gamma)$.*

Proof. By induction on the derivation of $\Gamma; \Delta \vdash M : T$:

Case (Val Var): assume that $\Gamma; \Delta \vdash y : T$ by the premise $(y : T) \in \Gamma$ with $\Gamma; \Delta \vdash \diamond$. We have $T = \psi(T)$ by Lemma 3.2.5, hence $forms(x : T) = \emptyset$ by Lemma 3.2.11 and we just need to show that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$ for some $!\Delta'$ such that $\Gamma; !\Delta' \vdash y : T$. Now we note that $\Gamma; \Delta \vdash \diamond$ implies $\Gamma; \emptyset \vdash \diamond$ by Lemma 3.2.5, hence $\Gamma; \emptyset \vdash y : T$ by (Val Var). This leads to the desired conclusion, since $\Gamma; \Delta \hookrightarrow \Gamma; \emptyset$ by Lemma 3.2.8.

Case (Val Unit): assume that $\Gamma; \Delta \vdash () : \mathsf{unit}$ by the premise $\Gamma; \Delta \vdash \diamond$. Since $\psi(\mathsf{unit}) = \mathsf{unit}$ and $forms(x : \mathsf{unit}) = \emptyset$, we just need to show that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$ for some $!\Delta'$ such that $\Gamma; !\Delta' \vdash () : \mathsf{unit}$. By Lemma 3.2.5 we have $\Gamma; \emptyset \vdash \diamond$, hence $\Gamma; \emptyset \vdash () : \mathsf{unit}$ by (Val Unit). This leads to the desired conclusion, since $\Gamma; \Delta \hookrightarrow \Gamma; \emptyset$ by Lemma 3.2.8.

Case (Val Fun): assume that $\Gamma; \Delta \vdash \lambda y. E : y : U_1 \to U_2$ by the premise $(\Gamma; !\Delta') \bullet y : U_1 \vdash E : U_2$ with $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$. Since $\psi(y : U_1 \to U_2) = y : U_1 \to U_2$ and $forms(x : (y : U_1 \to U_2)) = \emptyset$, the conclusion is immediate.

Case (Val Refine): assume that $\Gamma; \Delta \vdash M : \{x : U \mid F\}$ by the premises $\Gamma; \Delta_1 \vdash M : U$ and $\Gamma; \Delta_2 \vdash F\{M/x\}$ with $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$. Notice that $\Gamma; \Delta_2 \vdash F\{M/x\}$ implies $\Delta_2 \vdash F\{M/x\}$ by inverting rule (Derive). By inductive hypothesis $\Gamma; \Delta_1 \hookrightarrow \Gamma; !\Delta_{11}, \Delta_{12}$ with $\Gamma; !\Delta_{11} \vdash M : \psi(U)$ and $\Delta_{12} \vdash forms(x : U)\{M/x\}$. Notice that the former is equivalent to $\Gamma; !\Delta_{11} \vdash M : \psi(\{x : U \mid F\})$ by definition. Now by applying ($\otimes$-Right) we get $\Delta_{12}, \Delta_2 \vdash forms(x : U)\{M/x\} \otimes F\{M/x\}$, which is equivalent to $\Delta_{12}, \Delta_2 \vdash forms(x : \{x : U \mid F\})\{M/x\}$. Since $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_{11}, (\Delta_{12}, \Delta_2)$ by Lemma 3.2.8, we can conclude.

Case (Exp Subsum): assume that $\Gamma; \Delta \vdash M : T$ by the premises $\Gamma; \Delta_1 \vdash M : U$ and $\Gamma; \Delta_2 \vdash U <: T$ with $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$. By inductive hypothesis $\Gamma; \Delta_1 \hookrightarrow \Gamma; !\Delta_{11}, \Delta_{12}$ with $\Gamma; !\Delta_{11} \vdash M : \psi(U)$ and $\Delta_{12} \vdash forms(x : U)\{M/x\}$. By the premise $\Gamma; \Delta_2 \vdash U <: T$ and Lemma 3.2.15, we have $\Gamma; \Delta_2 \hookrightarrow \Gamma; !\Delta_{21}, \Delta_{22}$ with $\Gamma; !\Delta_{21} \vdash \psi(U) <: \psi(T)$ and $\Delta_{22}, forms(x : U) \vdash forms(x : T)$. Now we note that $x \notin dom(\Gamma)$ implies $x \notin fv(\Delta_{22})$ by Lemma 3.2.10. Given that logical entailment is closed under substitution of closed values for variables by Lemma 3.2.1, we get:

$$\Delta_{22}, forms(x : U)\{M/x\} \vdash forms(x : T)\{M/x\},$$

hence by Lemma 3.2.3 we have:

$$\Delta_{12}, \Delta_{22} \vdash forms(x : T)\{M/x\}.$$

Moreover, by (Exp Subsum) we get $\Gamma; !\Delta_{11}, !\Delta_{21} \vdash M : \psi(T)$, hence we conclude, since $\Gamma; \Delta \hookrightarrow \Gamma; (!\Delta_{11}, !\Delta_{21}), (\Delta_{12}, \Delta_{22})$ by Lemma 3.2.8.

The cases for rules (Val Pair), (Val Inl), (Val Inr) and (Val Fold) are identical to the case for (Val Fun). □

Lemma 3.2.23 ($\otimes$ Sub)**.** *If* $\Gamma; \Delta \vdash \{x : T \mid C_1 \otimes C_2\}$*, then* $\Gamma; \emptyset \vdash \{x : T \mid C_1 \otimes C_2\} <:> \{x : \{x : T \mid C_1\} \mid C_2\}$*.*

Proof. By applying (Sub Refine), using simple observations. □

Lemma 3.2.24 (Affine Typing)**.** *If* $\Gamma; \Delta \vdash M : T$*, then there exist* $!\Delta'$ *and* Δ'' *such that* $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta', \Delta''$ *and* $\Gamma; !\Delta' \vdash M : \psi(T)$ *and* $\Gamma; \Delta'' \vdash M : T$*.*

Proof. Let $\Gamma; \Delta \vdash M : T$ and consider any $x \notin \mathit{dom}(\Gamma)$. By Lemma 3.2.22 there exist $!\Delta'$ and Δ'' such that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta', \Delta''$ and $\Gamma; !\Delta' \vdash M : \psi(T)$ and $\Delta'' \vdash \mathit{forms}(x : T)\{M/x\}$. By Lemma 3.2.8 we have $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta', !\Delta', \Delta''$. Now we note that $\Gamma; !\Delta', \Delta'' \vdash M : \{x : \psi(T) \mid \mathit{forms}(x : T)\}$ by (Val Refine), hence $\Gamma; !\Delta', \Delta'' \vdash M : T$ by using (Exp Subsum) in combination with Lemma 3.2.23. Hence, we proved $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta', (!\Delta', \Delta'')$ with $\Gamma; !\Delta' \vdash M : \psi(T)$ and $\Gamma; !\Delta', \Delta'' \vdash M : T$. □

Lemma 3.2.25 (Formulas)**.** *If* $\Gamma; \Delta \vdash M : T$ *and* $x \notin \mathit{dom}(\Gamma)$*, then* $\Delta \vdash \mathit{forms}(x : T)\{M/x\}$*.*

Proof. Since $\Gamma; \Delta \vdash M : T$ and $x \notin \mathit{dom}(\Gamma)$, we apply Lemma 3.2.22 and we get that there exist $!\Delta', \Delta''$ such that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta', \Delta''$ and $\Delta'' \vdash \mathit{forms}(x : T)\{M/x\}$. By inverting (Rewrite) we know that $\Delta \vdash !\Delta', \Delta''$. By multiple applications of (Weak) we get $!\Delta', \Delta'' \vdash \mathit{forms}(x : T)\{M/x\}$, hence $\Delta \vdash \mathit{forms}(x : T)\{M/x\}$ by Lemma 3.2.3. □

Lemma 3.2.26 (Basic Substitution)**.** *The following statements hold true:*

1. *For every type* T*, we have* $\psi(T)\{M/x\} = \psi(T\{M/x\})$*.*
2. *If* $x \neq y$*, then* $\mathit{forms}(y : T)\{M/x\} = \mathit{forms}(y : T\{M/x\})$*.*

Proof. Point (1) is proved by induction on the structure of T, while point (2) follows by definition of *forms* and standard syntactic properties of substitution. □

Lemma 3.2.27 (Substitution)**.** *Suppose that* $\Gamma; \Delta \vdash M : U$ *and* $\mathit{fv}(M) = \emptyset$*. The following statements hold true:*

1. *If* $(\Gamma; \Delta') \bullet x : U \bullet (\Gamma'; \Delta'') \vdash \diamond$*, then*
   $\Gamma, \Gamma'\{M/x\}; \Delta, (\Delta', \Delta'')\{M/x\} \vdash \diamond$*.*
2. *If* $(\Gamma; \Delta') \bullet x : U \bullet (\Gamma'; \Delta'') \vdash F$*, then*
   $\Gamma, \Gamma'\{M/x\}; \Delta, (\Delta', \Delta'')\{M/x\} \vdash F\{M/x\}$*.*

3. *If* $(\Gamma;\Delta')\bullet x:U\bullet(\Gamma';\Delta'')\hookrightarrow\Gamma,x:\psi(U),\Gamma';\Delta^*$, *then* $\Gamma,\Gamma'\{M/x\};\Delta,(\Delta',\Delta'')\{M/x\}\hookrightarrow\Gamma,\Gamma'\{M/x\};\Delta^*\{M/x\}$.

4. *If* $(\Gamma;\Delta')\bullet x:U\bullet(\Gamma';\Delta'')\vdash T$, *then* $\Gamma,\Gamma'\{M/x\};\Delta,(\Delta',\Delta'')\{M/x\}\vdash T\{M/x\}$.

5. *If* $(\Gamma;\Delta')\bullet x:U\bullet(\Gamma';\Delta'')\vdash T::k$, *then* $\Gamma,\Gamma'\{M/x\};\Delta,(\Delta',\Delta'')\{M/x\}\vdash T\{M/x\}::k$.

6. *If* $(\Gamma;\Delta')\bullet x:U\bullet(\Gamma';\Delta'')\vdash T<:T'$, *then* $\Gamma,\Gamma'\{M/x\};\Delta,(\Delta',\Delta'')\{M/x\}\vdash T\{M/x\}<:T'\{M/x\}$.

7. *If* $(\Gamma;\Delta')\bullet x:U\bullet(\Gamma';\Delta'')\vdash E:T$, *then* $\Gamma,\Gamma'\{M/x\};\Delta,(\Delta',\Delta'')\{M/x\}\vdash E\{M/x\}:T\{M/x\}$.

Proof. The proof is by simultaneous induction on the derivation of the antecedent judgements:

1. Rule (Env Empty) cannot be applied. For the other two rules the conclusion follows by inductive hypothesis, using Lemma 3.2.5 and standard syntactic properties of substitution.

2. Let $(\Gamma;\Delta')\bullet x:U\bullet(\Gamma';\Delta'')\vdash F$. The previous typing environment is equivalent to $\Gamma,x:\psi(U),\Gamma';\Delta',\mathit{forms}(x:U),\Delta''$, thus by inverting rule (Derive) we have $(\Gamma;\Delta')\bullet x:U\bullet(\Gamma';\Delta'')\vdash\diamond$ and $\mathit{fnfv}(F)\subseteq\mathit{dom}(\Gamma,x:\psi(U),\Gamma')$ and $\Delta',\mathit{forms}(x:U),\Delta''\vdash F$.

   By inductive hypothesis we have $\Gamma,\Gamma'\{M/x\};\Delta,(\Delta',\Delta'')\{M/x\}\vdash\diamond$. Since $\Gamma;\Delta\vdash M:U$ implies $\mathit{fnfv}(M)\subseteq\mathit{dom}(\Gamma)$ by Lemma 3.2.5, it is easy to observe that $\mathit{fnfv}(F\{M/x\})\subseteq\mathit{dom}(\Gamma,\Gamma'\{M/x\})$. Given that logical entailment is closed under substitution of closed values for variables by Lemma 3.2.1, we have:
   $$(\Delta',\mathit{forms}(x:U),\Delta'')\{M/x\}\vdash F\{M/x\}.$$

   Now we note that by Lemma 3.2.25 we have $\Delta\vdash\mathit{forms}(x:U)\{M/x\}$, hence $\Delta,(\Delta',\Delta'')\{M/x\}\vdash F\{M/x\}$ by Lemma 3.2.3.

   The conclusion $\Gamma,\Gamma'\{M/x\};\Delta,(\Delta',\Delta'')\{M/x\}\vdash F\{M/x\}$ then follows by applying (Derive).

3. Let $(\Gamma;\Delta')\bullet x:U\bullet(\Gamma';\Delta'')\hookrightarrow\Gamma,x:\psi(U),\Gamma';\Delta^*$. We first note that the environment $(\Gamma;\Delta')\bullet x:U\bullet(\Gamma';\Delta'')$ is equivalent to $\Gamma,x:\psi(U),\Gamma';\Delta',\mathit{forms}(x:U),\Delta''$, thus by inverting rule (Rewrite) we have $(\Gamma;\Delta')\bullet x:U\bullet(\Gamma';\Delta'')\vdash\diamond$ and $\Delta',\mathit{forms}(x:U),\Delta''\vdash\Delta^*$ and $(\Gamma;\emptyset)\bullet x:\psi(U)\bullet(\Gamma';\Delta^*)\vdash\diamond$.

   We apply the inductive hypothesis to $(\Gamma;\Delta')\bullet x:U\bullet(\Gamma';\Delta'')\vdash\diamond$ and we get $\Gamma,\Gamma'\{M/x\};\Delta,(\Delta',\Delta'')\{M/x\}\vdash\diamond$.

Given that logical entailment is closed under substitution of closed values for variables by Lemma 3.2.1, we have:

$$(\Delta', \mathit{forms}(x : U), \Delta'')\{M/x\} \vdash \Delta^*\{M/x\},$$

Now we note that by Lemma 3.2.25 we have $\Delta \vdash \mathit{forms}(x : U)\{M/x\}$, hence $\Delta, (\Delta', \Delta'')\{M/x\} \vdash \Delta^*\{M/x\}$ by Lemma 3.2.3.

By Lemma 3.2.24 we have $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, \Delta_2$ for some $!\Delta_1, \Delta_2$ such that $\Gamma; !\Delta_1 \vdash M : \psi(U)$ and $\Gamma; \Delta_2 \vdash M : U$. We then apply the inductive hypothesis to $(\Gamma; \emptyset) \bullet x : \psi(U) \bullet (\Gamma'; \Delta^*) \vdash \diamond$ and we get $\Gamma, \Gamma'\{M/x\}; !\Delta_1, \Delta^*\{M/x\} \vdash \diamond$. By Lemma 3.2.5 this implies $\Gamma, \Gamma'\{M/x\}; \Delta^*\{M/x\} \vdash \diamond$, hence we conclude by applying (Rewrite).

4. We just need to consider rule (Type). The conclusion follows by inverting the rule, using point (1), Lemma 3.2.5 and standard syntactic properties of substitution.

5. Rules (Kind Var) and (Kind Unit) use point (1). Rule (Kind Refine Public) uses point (3) and the inductive hypothesis. The rules involving both logical rewriting and splitting are the most interesting, we show (Kind Pair) as an example.

   Assume then that $(\Gamma; \Delta') \bullet x : U \bullet (\Gamma'; \Delta'') \vdash y : T_1 * T_2 :: k$ by the premises $\Gamma, x : \psi(U), \Gamma'; !\Delta_1 \vdash T_1 :: k$ and $\Gamma, x : \psi(U), \Gamma', y : \psi(T_1); !\Delta_2 \vdash T_2 :: k$ with $(\Gamma; \Delta') \bullet x : U \bullet (\Gamma'; \Delta'') \hookrightarrow \Gamma, x : \psi(U), \Gamma'; !\Delta_1, !\Delta_2$.

   We note that we can state the two premises as $(\Gamma; !\Delta_1) \bullet x : \psi(U) \bullet (\Gamma'; \emptyset) \vdash T_1 :: k$ and $(\Gamma; !\Delta_2) \bullet x : \psi(U) \bullet (\Gamma', y : \psi(T_1); \emptyset) \vdash T_2 :: k$. By Lemma 3.2.24 in combination with Lemma 3.2.8 we have that $\Gamma; \Delta \vdash M : U$ implies $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'_1, !\Delta'_1, \Delta'_2$ for some $!\Delta'_1$ and Δ'_2 such that $\Gamma; !\Delta'_1 \vdash M : \psi(U)$ and $\Gamma; \Delta'_2 \vdash M : U$.

   We now apply the inductive hypothesis twice, and get:

$$\Gamma, \Gamma'\{M/x\}; !\Delta'_1, !\Delta_1\{M/x\} \vdash T_1\{M/x\} :: k,$$

   and:

$$\Gamma, \Gamma'\{M/x\}, y : \psi(T_1)\{M/x\}; !\Delta'_1, !\Delta_2\{M/x\} \vdash T_2\{M/x\} :: k.$$

   Notice that, by Lemma 3.2.26, the latter is equivalent to:

$$\Gamma, \Gamma'\{M/x\}, y : \psi(T_1\{M/x\}); !\Delta'_1, !\Delta_2\{M/x\} \vdash T_2\{M/x\} :: k.$$

   We then proceed by considering the premise $(\Gamma; \Delta') \bullet x : U \bullet (\Gamma'; \Delta'') \hookrightarrow \Gamma, x : \psi(U), \Gamma'; !\Delta_1, !\Delta_2$. We apply the inductive hypothesis (point 3) there and we get $\Gamma, \Gamma'\{M/x\}; \Delta'_2, (\Delta', \Delta'')\{M/x\} \hookrightarrow \Gamma, \Gamma'\{M/x\}; (!\Delta_1, !\Delta_2)\{M/x\}$.

Now we note that:

$\Gamma, \Gamma'\{M/x\}; \Delta, (\Delta', \Delta'')\{M/x\} \hookrightarrow \Gamma, \Gamma'\{M/x\}; (!\Delta'_1, !\Delta_1\{M/x\}), (!\Delta'_1, !\Delta_2\{M/x\})$,

hence we can conclude by applying (Kind Pair).

6. Rule (Sub Refl) uses point (4). Rule (Sub Pub Tnt) uses point (5). The remaining cases mostly rely on the same arguments applied to prove the case (Kind Pair) of the previous point. We show (Sub Refine) as an example case.

   Assume then that $(\Gamma; \Delta') \bullet x : U \bullet (\Gamma'; \Delta'') \vdash T_1 <: T_2$ by the premises $\Gamma, x : \psi(U), \Gamma'; \Delta_1 \vdash \psi(T_1) <: \psi(T_2)$ and $(\Gamma, x : \psi(U), \Gamma'; \Delta_2) \bullet y : T_1 \vdash \mathit{forms}(y : T_2)$ with $(\Gamma; \Delta') \bullet x : U \bullet (\Gamma'; \Delta'') \hookrightarrow \Gamma, x : \psi(U), \Gamma'; \Delta_1, \Delta_2$.

   We note that we can state the two premises as $(\Gamma; \Delta_1) \bullet x : \psi(U) \bullet (\Gamma'; \emptyset) \vdash \psi(T_1) <: \psi(T_2)$ and $(\Gamma; \Delta_2) \bullet x : \psi(U) \bullet (\Gamma', y : \psi(T_1); \mathit{forms}(y : T_1)) \vdash \mathit{forms}(y : T_2)$. By Lemma 3.2.24 in combination with Lemma 3.2.8 we have that $\Gamma; \Delta \vdash M : U$ implies $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'_1, !\Delta'_1, \Delta'_2$ for some $!\Delta'_1$ and Δ'_2 such that $\Gamma; !\Delta'_1 \vdash M : \psi(U)$ and $\Gamma; \Delta'_2 \vdash M : U$.

   We now apply the inductive hypothesis twice, and get:

   $$\Gamma, \Gamma'\{M/x\}; !\Delta'_1, \Delta_1\{M/x\} \vdash \psi(T_1)\{M/x\} <: \psi(T_2)\{M/x\},$$

   and:

   $\Gamma, \Gamma'\{M/x\}, y : \psi(T_1)\{M/x\}; !\Delta'_1, (\Delta_2, \mathit{forms}(y : T_1))\{M/x\} \vdash \mathit{forms}(y : T_2)\{M/x\}$.

   By Lemma 3.2.26, the former is equivalent to:

   $$\Gamma, \Gamma'; !\Delta'_1, \Delta_1\{M/x\} \vdash \psi(T_1\{M/x\}) <: \psi(T_2\{M/x\}),$$

   while the latter is equivalent to:

   $\Gamma, \Gamma', y : \psi(T_1\{M/x\}); !\Delta'_1, \Delta_2\{M/x\}, \mathit{forms}(y : T_1\{M/x\}) \vdash \mathit{forms}(y : T_2\{M/x\})$.

   We then proceed by considering the premise $(\Gamma; \Delta') \bullet x : U \bullet (\Gamma'; \Delta'') \hookrightarrow \Gamma, x : \psi(U), \Gamma'; \Delta_1, \Delta_2$. We apply the inductive hypothesis (point 3) there and we get $\Gamma, \Gamma'\{M/x\}; \Delta'_2, (\Delta', \Delta'')\{M/x\} \hookrightarrow \Gamma, \Gamma'\{M/x\}; (\Delta_1, \Delta_2)\{M/x\}$.

   Now we note that:

   $\Gamma, \Gamma'\{M/x\}; \Delta, (\Delta', \Delta'')\{M/x\} \hookrightarrow \Gamma, \Gamma'\{M/x\}; (!\Delta'_1, \Delta_1\{M/x\}), (!\Delta'_1, \Delta_2\{M/x\})$,

   hence we can conclude by applying (Sub Refine).

7. All cases follow by the previous points and the inductive hypothesis, using standard syntactic properties of substitution and replicating the same arguments as before.

□

3.2.5 Proofs of inversion lemmas

Lemma 3.2.28 (Bound Weak). *Let* $\Gamma; \Delta \vdash T' <: T$*. If* $\Gamma, x : \psi(T), \Gamma'; \Delta', \mathit{forms}(x : T) \vdash \mathcal{J}$*, then* $\Gamma, x : \psi(T'), \Gamma'; \Delta, \Delta', \mathit{forms}(x : T') \vdash \mathcal{J}$*.*

Proof. For each judgement $\mathcal{J}$ the proof proceeds by induction on the derivation of $\Gamma, x : \psi(T), \Gamma'; \Delta', \mathit{forms}(x : T) \vdash \mathcal{J}$. We frequently use the fact that $\mathit{dom}(\Gamma, x : \psi(T), \Gamma') = \mathit{dom}(\Gamma, x : \psi(T'), \Gamma')$. Furthermore, we often use Lemma 3.2.7 implicitly.

1. $\mathcal{J} = \diamond$: The induction proof uses the fact that by Lemma 3.2.5 we know that $\mathit{fnfv}(T') \subseteq \mathit{dom}(\Gamma)$ and $\mathit{fnfv}(\Delta) \subseteq \mathit{dom}(\Gamma)$.

2. $\mathcal{J} = U$: By Lemma 3.2.5 we know that $\mathit{fnfv}(U) \subseteq \mathit{dom}(\Gamma, x : \psi(T), \Gamma')$, which means that also $\mathit{fnfv}(U) \subseteq \mathit{dom}(\Gamma, x : \psi(T'), \Gamma')$. We conclude by applying statement (1) and rule (TYPE).

3. $\mathcal{J} = F$: The proof makes use of statement (1), Lemma 3.2.5, and the fact that $\mathit{dom}(\Gamma, x : \psi(T), \Gamma') = \mathit{dom}(\Gamma, x : \psi(T'), \Gamma')$. Furthermore, it applies Lemma 3.2.15 to $\Gamma; \Delta \vdash T' <: T$ (showing that formulas in Δ and T' entail those in T) in combination with Lemma 3.2.3 to conclude.

4. $\mathcal{J} = U :: k$: The proof makes uses of the previous statements. It also uses Lemma 3.2.16 to show that replacing $x : \psi(T)$ by $x : \psi(T')$ is safe. It applies Lemma 3.2.15 to $\Gamma; \Delta \vdash T' <: T$ (showing that formulas in Δ and T' entail those in T) in combination with Lemma 3.2.3 to conclude.

5. $\mathcal{J} = U <: U'$: The proof uses similar reasoning as the proof of statement (4) and makes use of the previous statements.

6. $\mathcal{J} = E : U$: The proof makes use of the previous statements and relies on Lemma 3.2.15 and Lemma 3.2.3.

□

Lemma 3.2.29 (Type Variables and Kinding). *If* $\Gamma, \alpha, \Gamma'; \Delta \vdash T :: k$*, then* $\alpha \notin \mathit{fnfv}(T)$*.*

Proof. By induction on the derivation of $\Gamma, \alpha, \Gamma'; \Delta \vdash T :: k$.

The case (KIND UNIT) follows immediately, since $\mathit{fnfv}(\mathsf{unit}) = \emptyset$. The case (KIND VAR) implies that $T = \beta$ for some type variable β and $\beta :: k \in (\Gamma, \alpha, \Gamma')$. It must be the case that $\beta \neq \alpha$, since $\Gamma, \alpha, \Gamma'; \Delta \vdash T :: k$ implies $\Gamma, \alpha, \Gamma'; \Delta \vdash \diamond$ by Lemma 3.2.5 and having both α and $\alpha :: k$ in the same environment would violate the well-formedness conditions enforced by (TYPE ENV ENTRY), given that $\mathit{dom}(\alpha) = \mathit{dom}(\alpha :: k)$. The case (KIND REFINE TAINTED) follows by an application of the induction hypothesis to the first premise of the kinding rule, using the fact that $\alpha \in \mathit{fnfv}(T)$ if and only if $\alpha \in \mathit{fnfv}(\psi(T))$. The remaining cases follow by the induction hypothesis. □

Lemma 3.2.30 (Type Substitution). *For all T, T' such that $T = \psi(T)$ and $T' = \psi(T')$ it holds that:*

1. *If $\Gamma, \alpha, \Gamma'; \Delta \vdash \mathcal{J}$ and $\Gamma; \Delta' \vdash T$, then $\Gamma, (\Gamma'\{T/\alpha\}); \Delta, \Delta' \vdash \mathcal{J}\{T/\alpha\}$.*
2. *If $\Gamma, \alpha :: k, \Gamma'; \Delta \vdash \diamond$ and $\Gamma; \Delta' \vdash T :: k$, then $\Gamma, (\Gamma'\{T/\alpha\}); \Delta, \Delta' \vdash \diamond$.*
3. *If $\Gamma, \alpha :: k, \Gamma'; \Delta \vdash U$ and $\Gamma; \Delta' \vdash T :: k$, then $\Gamma, (\Gamma'\{T/\alpha\}); \Delta, \Delta' \vdash U\{T/\alpha\}$.*
4. *If $\Gamma, \alpha :: k, \Gamma'; \Delta \vdash U :: k'$ and $\Gamma; \Delta' \vdash T :: k$, then $\Gamma, (\Gamma'\{T/\alpha\}); \Delta, \Delta' \vdash U\{T/\alpha\} :: k'$.*
5. *We have:*
   - *If $\Gamma, \alpha, \Gamma'; \emptyset \vdash U$ and α only occurs positively in U and $\Gamma; !\Delta \vdash T <: T'$, then $\Gamma, (\Gamma'\{T/\alpha\}); !\Delta \vdash U\{T/\alpha\} <: U\{T'/\alpha\}$.*
   - *If $\Gamma, \alpha, \Gamma'; \emptyset \vdash U$ and α only occurs negatively in U and $\Gamma; !\Delta \vdash T <: T'$, then $\Gamma, (\Gamma'\{T/\alpha\}); !\Delta \vdash U\{T'/\alpha\} <: U\{T/\alpha\}$.*
6. *If $\Gamma, \alpha, \Gamma'; \Delta \vdash U <: U'$ and α only occurs positively in U, U' and $\Gamma; \Delta' \vdash T <: T'$, then $\Gamma, (\Gamma'\{T/\alpha\}); \Delta, \Delta' \vdash U\{T/\alpha\} <: U'\{T'/\alpha\}$.*

Proof. We would like to note that the core statements of this lemma are points (4) and (6), the other points are just needed to prove them. In particular, point (1) is often used in the proof of the later statements; point (2) is used in the proof of point (3), which in turn is used in the proof of point (4); point (5) is used in the proof of point (6). We provide a proof sketch below.

1. By induction on the derivation of $\Gamma, \alpha, \Gamma'; \Delta \vdash \mathcal{J}$, making use of Lemma 3.2.5 and Lemma 3.2.6.

2. By induction on the derivation of $\Gamma, \alpha :: k, \Gamma'; \Delta \vdash \diamond$, making use of Lemma 3.2.5 and Lemma 3.2.6.

3. By the definition of (Type), using point (2), Lemma 3.2.5 and Lemma 3.2.6.

4. Since we know that $T = \psi(T)$, we can apply Lemma 3.2.13 in combination with Lemma 3.2.8 to show that there exists a Δ'' such that $\Gamma; \Delta' \hookrightarrow \Gamma; !\Delta''$ and $\Gamma; !\Delta'' \vdash T :: k$.

   We now prove the following modified statement: if $\Gamma, \alpha :: k, \Gamma'; \Delta \vdash U :: k'$ and $\Gamma; !\Delta'' \vdash T :: k$, then $\Gamma, (\Gamma'\{T/\alpha\}); \Delta, !\Delta'' \vdash U\{T/\alpha\} :: k'$.

   The proof proceeds by induction on the derivation of $\Gamma, \alpha :: k, \Gamma'; \Delta \vdash U :: k'$, using point (3) and making use of Lemma 3.2.8 whenever needed to perform the rewriting $\Gamma; !\Delta'' \hookrightarrow \Gamma; !\Delta'', !\Delta''$ and apply the inductive hypothesis twice.

   The conclusion then follows by Lemma 3.2.9.

5. We prove both points simultaneously by induction on the structure of U.

6. Since we know that $T = \psi(T)$ and $T' = \psi(T')$, we can apply Lemma 3.2.15 in combination with Lemma 3.2.8 to show that there exists a Δ'' such that $\Gamma; \Delta' \hookrightarrow \Gamma; !\Delta''$ and $\Gamma; !\Delta'' \vdash T <: T'$.

   We now prove the following modified statement: if $\Gamma, \alpha, \Gamma'; \Delta \vdash U <: U'$ and α only occurs positively in U, U' and $\Gamma; !\Delta'' \vdash T <: T'$, then $\Gamma, (\Gamma'\{T/\alpha\}); \Delta, !\Delta'' \vdash U\{T/\alpha\} <: U'\{T'/\alpha\}$.

   The proof proceeds by induction on the derivation of $\Gamma, \alpha, \Gamma'; \Delta \vdash U <: U'$, using point (5) and making use of Lemma 3.2.8 whenever needed to perform the rewriting $\Gamma; !\Delta'' \hookrightarrow \Gamma; !\Delta'', !\Delta''$ and apply the inductive hypothesis twice.

   The conclusion then follows by Lemma 3.2.9.

□

Lemma 3.2.31 (Inversion for Functions). *The following statements hold:*

1. *If $\Gamma; \Delta \vdash \lambda x. E : V$, then there exist Δ_1, Δ_2, T, U such that $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$ and $\Gamma; \Delta_1 \vdash \lambda x. E : x : T \to U$ (by a top-level application of* Val Fun*) and $\Gamma; \Delta_2 \vdash x : T \to U <: \psi(V)$.*

2. *If $\Gamma; \Delta \vdash x : T \to U <: x : T' \to U'$, then there exist Δ_1, Δ_2 such that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2$ and $\Gamma; !\Delta_1 \vdash T' <: T$ and $\Gamma, x : \psi(T'); !\Delta_2 \vdash U <: U'$.*

3. *If $\Gamma; \Delta \vdash \lambda x. E : x : T \to U$, then there exists a Δ' such that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$ and $(\Gamma; !\Delta') \bullet x : T \vdash E : U$.*

4. *If $\Gamma; \Delta \vdash \lambda x. E : x : T \to U$, then $(\Gamma; \Delta) \bullet x : T \vdash E : U$.*

Proof. We show the four statements separately, using the first two results in the proof of the third.

1. By induction on the derivation of $\Gamma; \Delta \vdash \lambda x. E : V$. We know that $\Gamma; \Delta \vdash \lambda x. E : V$. We distinguish three cases, depending on the last applied typing rule:

Case (Val Fun): In this case we know that $V = x : T \to U$ for some T, U, hence $\psi(V) = V$. Since $\Delta; \emptyset \vdash \psi(V)$ by Lemma 3.2.5, we immediately derive $\Gamma; \emptyset \vdash x : T \to U <: \psi(V)$ by (Sub Refl). Since $\Gamma; \Delta \hookrightarrow \Gamma; \Delta$ by Lemma 3.2.8, we can conclude.

Case (Val Refine): In this case we know that $V = \{y : V' \mid F\}$ and $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$ for some Δ_1, Δ_2 such that $\Gamma; \Delta_1 \vdash \lambda x. E : V'$ and $\Gamma; \Delta_2 \vdash F\{\lambda x. E/y\}$. We can apply the induction hypothesis to $\Gamma; \Delta_1 \vdash \lambda x. E : V'$, letting us derive that there exist $\Delta_{11}, \Delta_{12}, T, U$ such that:

- $\Gamma; \Delta_1 \hookrightarrow \Gamma; \Delta_{11}, \Delta_{12}$,
- $\Gamma; \Delta_{11} \vdash \lambda x. E : x : T \to U$ by a top-level application of (VAL FUN), and
- $\Gamma; \Delta_{12} \vdash x : T \to U <: \psi(V')$.

By the definition of ψ we know that $\psi(V) = \psi(V')$, thus we know that:

- $\Gamma; \Delta_{11} \vdash \lambda x. E : x : T \to U$ by a top-level application of (VAL FUN), and
- $\Gamma; \Delta_{12} \vdash x : T \to U <: \psi(V)$.

Since $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_{11}, \Delta_{12}$ by Lemma 3.2.8, we can conclude.

Case (EXP SUBSUM): In this case we know that there exist Δ_1, Δ_2 such that $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$ and $\Gamma; \Delta_1 \vdash \lambda x. E : V'$ and $\Gamma; \Delta_2 \vdash V' <: V$.

We can apply the induction hypothesis to $\Gamma; \Delta_1 \vdash \lambda x. E : V'$, letting us derive that there exist $\Delta_{11}, \Delta_{12}, T, U$ such that:

- $\Gamma; \Delta_1 \hookrightarrow \Gamma; \Delta_{11}, \Delta_{12}$,
- $\Gamma; \Delta_{11} \vdash \lambda x. E : x : T \to U$ by a top-level application of (VAL FUN), and
- $\Gamma; \Delta_{12} \vdash x : T \to U <: \psi(V')$.

We apply Lemma 3.2.15 to $\Gamma; \Delta_2 \vdash V' <: V$ and we get that there exist $!\Delta_{21}, \Delta_{22}$ such that $\Gamma; \Delta_2 \hookrightarrow \Gamma; !\Delta_{21}, \Delta_{22}$ and $\Gamma; !\Delta_{21} \vdash \psi(V') <: \psi(V)$. Since $\Gamma; \Delta_2 \hookrightarrow \Gamma; !\Delta_{21}$ by Lemma 3.2.8 point 1, we have $\Gamma; \Delta_2 \vdash \psi(V') <: \psi(V)$ by Lemma 3.2.9. By transitivity of the subtyping relation (Lemma 3.2.21) we thus have:

$$\Gamma; \Delta_{12}, \Delta_2 \vdash x : T \to U <: \psi(V),$$

which allows us to conclude.

2. By induction on the derivation of $\Gamma; \Delta \vdash x : T \to U <: x : T' \to U'$. We implicitly use Lemma 3.2.5 whenever needed. We distinguish three cases, depending on the last applied subtyping rule:

Case (SUB REFL): In this case we know that $T = T'$ and $U = U'$ and conclude by two applications of (SUB REFL) that $\Gamma; \emptyset \vdash T' <: T$ and $\Gamma; \emptyset \vdash U <: U'$. Using suitable alpha-renaming and Lemma 3.2.7 to extend Γ with $x : \psi(T')$ in the second judgement, we can conclude, since $\Gamma; \Delta \hookrightarrow \Gamma; \emptyset$ by Lemma 3.2.8.

Case (SUB FUN): The statement follows immediately by the premises of the subtyping rule.

Case (SUB PUB TNT): In this case we know that $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$ for some Δ_1, Δ_2 such that $\Gamma; \Delta_1 \vdash x : T \to U :: \mathsf{pub}$ and $\Gamma; \Delta_2 \vdash x : T' \to U' :: \mathsf{tnt}$.

By the only applicable kinding rule (KIND FUN) it follows that there exist Δ_{11}, Δ_{12} and Δ_{21}, Δ_{22} such that $\Gamma; \Delta_1 \hookrightarrow \Gamma; \Delta_{11}, \Delta_{12}$ and $\Gamma; \Delta_2 \hookrightarrow \Gamma; \Delta_{21}, \Delta_{22}$ such that $\Gamma; !\Delta_{11} \vdash T :: \mathsf{tnt}$ and $\Gamma; !\Delta_{21} \vdash T' :: \mathsf{pub}$ and $\Gamma, x : \psi(T); !\Delta_{12} \vdash U :: \mathsf{pub}$ and $\Gamma, x : \psi(T'); !\Delta_{22} \vdash U' :: \mathsf{tnt}$.

Applying (Sub Pub Tnt) to $\Gamma; !\Delta_{11} \vdash T :: \mathsf{tnt}$ and $\Gamma; !\Delta_{21} \vdash T' :: \mathsf{pub}$ yields:

$$\Gamma; !\Delta_{11}, !\Delta_{21} \vdash T' <: T.$$

We apply Lemma 3.2.16 to $\Gamma, x : \psi(T); !\Delta_{12} \vdash U :: \mathsf{pub}$ and we get:

$$\Gamma, x : \psi(T'); !\Delta_{12} \vdash U :: \mathsf{pub}.$$

Applying (Sub Pub Tnt) to $\Gamma, x : \psi(T'); !\Delta_{12} \vdash U :: \mathsf{pub}$ and $\Gamma, x : \psi(T'); !\Delta_{22} \vdash U' :: \mathsf{tnt}$ yields:

$$\Gamma, x : \psi(T'); !\Delta_{12}, !\Delta_{22} \vdash U <: U',$$

thus allowing us to conclude.

3. We know that $\Gamma; \Delta \vdash \lambda x. E : x : T \to U$ and $\psi(x : T \to U) = x : T \to U$ by definition. We apply part (1) and derive that there exist $\Delta_1, \Delta_2, T', U'$ such that:

   - $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$,
   - $\Gamma; \Delta_1 \vdash \lambda x. E : x : T' \to U'$ by a top-level application of (Val Fun), and
   - $\Gamma; \Delta_2 \vdash x : T' \to U' <: x : T \to U$.

   By the definition of (Val Fun) the second statement lets us derive that:

   $$\Gamma, x : \psi(T'); !\Delta_1', \mathit{forms}(x : T') \vdash E : U',$$

   for some Δ_1' such that $\Gamma; \Delta_1 \hookrightarrow \Gamma; !\Delta_1'$, which is equivalent to $(\Gamma; !\Delta_1') \bullet x : T' \vdash E : U'$.

   Applying part (2) to the third statement yields that there exist Δ_{21}, Δ_{22} such that:

   - $\Gamma; \Delta_2 \hookrightarrow \Gamma; !\Delta_{21}, !\Delta_{22}$,
   - $\Gamma; !\Delta_{21} \vdash T <: T'$, and
   - $\Gamma, x : \psi(T); !\Delta_{22} \vdash U' <: U$.

   Applying Lemma 3.2.16 to the latter yields:

   $$\Gamma, x : \psi(T'); !\Delta_{22} \vdash U' <: U.$$

   We apply (Exp Subsum) to $(\Gamma; !\Delta_1') \bullet x : T' \vdash E : U'$ and $\Gamma, x : \psi(T'); !\Delta_{22} \vdash U' <: U$, which leads to:

   $$(\Gamma; !\Delta_1', !\Delta_{22}) \bullet x : T' \vdash E : U.$$

   Applying Lemma 3.2.28 to the latter statement and $\Gamma; !\Delta_{21} \vdash T <: T'$ lets us derive:

   $$(\Gamma; !\Delta_1', !\Delta_{22}, !\Delta_{21}) \bullet x : T \vdash E : U.$$

   Since $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1', !\Delta_{22}, !\Delta_{21}$ by Lemma 3.2.8, we can conclude.

4. Follows immediately from statement (3) by an application of Lemma 3.2.9.

□

Lemma 3.2.32 (Inversion for Pairs)**.** *The following statements hold:*

1. *If* $\Gamma; \Delta \vdash (M, N) : V$*, then there exist* Δ_1, Δ_2, T, U *such that* $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$ *and* $\Gamma; \Delta_1 \vdash (M, N) : x : T * U$ *(by a top-level application of* Val Pair*) and* $\Gamma; \Delta_2 \vdash x : T * U <: \psi(V)$*.*

2. *If* $\Gamma; \Delta \vdash x : T * U <: x : T' * U'$*, then there exist* Δ_1, Δ_2 *such that* $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2$ *and* $\Gamma; !\Delta_1 \vdash T <: T'$ *and* $\Gamma, x : \psi(T); !\Delta_2 \vdash U <: U'$*.*

3. *If* $\Gamma; \Delta \vdash (M, N) : x : T * U$*, then there exist* Δ_1, Δ_2 *such that* $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2$ *and* $\Gamma; !\Delta_1 \vdash M : T$ *and* $\Gamma; !\Delta_2 \vdash N : U\{M/x\}$*.*

Proof. We show the three statements separately, using the first two results in the proof of the third.

1. By induction on the derivation of $\Gamma; \Delta \vdash (M, N) : V$. The proof is analogous to that of Lemma 3.2.31, part (1).

2. By induction on the derivation of $\Gamma; \Delta \vdash x : T * U <: x : T' * U'$. The proof is analogous to that of Lemma 3.2.31, part (2).

3. We know that $\Gamma; \Delta \vdash (M, N) : x : T * U$ and that $\psi(x : T * U) = x : T * U$. We apply part (1) and derive that there exist $\Delta_1, \Delta_2, T', U'$ such that:

   - $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$,
   - $\Gamma; \Delta_1 \vdash (M, N) : x : T' * U'$ by a top-level application of (Val Pair), and
   - $\Gamma; \Delta_2 \vdash x : T' * U' <: x : T * U$.

   By the definition of (Val Pair) the second statement lets us derive:

   $$\Gamma; !\Delta_{11} \vdash M : T',$$

   and:

   $$\Gamma; !\Delta_{12} \vdash N : U'\{M/x\},$$

   for some Δ_{11}, Δ_{12} such that $\Gamma; \Delta_1 \hookrightarrow \Gamma; !\Delta_{11}, !\Delta_{12}$.

   We can also apply part (2) to the third statement, which let us derive that there exist Δ_{21}, Δ_{22} such that:

   - $\Gamma; \Delta_2 \hookrightarrow \Gamma; !\Delta_{21}, !\Delta_{22}$,
   - $\Gamma; !\Delta_{21} \vdash T' <: T$, and
   - $\Gamma, x : \psi(T'); !\Delta_{22} \vdash U' <: U$.

We apply (Exp Subsum) to $\Gamma; !\Delta_{11} \vdash M : T'$ and $\Gamma; !\Delta_{21} \vdash T' <: T$, which yields:

$$\Gamma; !\Delta_{11}, !\Delta_{21} \vdash M : T.$$

We know that $\Gamma, x : \psi(T'); !\Delta_{22} \vdash U' <: U$, which by applying Lemma 3.2.7 implies that $(\Gamma; !\Delta_{22}) \bullet x : T' \vdash U' <: U$ (we implicitly use the definition of "$\bullet$").

Since $\Gamma; !\Delta_{11} \vdash M : T'$, we can apply Lemma 3.2.27 to the latter statement and derive:

$$\Gamma; !\Delta_{11}, (!\Delta_{22}\{M/x\}) \vdash U'\{M/x\} <: U\{M/x\}.$$

Note, however, that since $x \notin \mathit{dom}(\Gamma)$ and $\Gamma; \Delta_2 \hookrightarrow \Gamma; !\Delta_{21}, !\Delta_{22}$, we know that $x \notin \mathit{fv}(\Delta_{22})$ by Lemma 3.2.10. Thus, the previous judgement is equivalent to:

$$\Gamma; !\Delta_{11}, !\Delta_{22} \vdash U'\{M/x\} <: U\{M/x\}.$$

We apply (Exp Subsum) to $\Gamma; !\Delta_{12} \vdash N : U'\{M/x\}$ and $\Gamma; !\Delta_{11}, !\Delta_{22} \vdash U'\{M/x\} <: U\{M/x\}$, which leads to:

$$\Gamma; !\Delta_{12}, !\Delta_{11}, !\Delta_{22} \vdash N : U\{M/x\}.$$

Using Lemma 3.2.8 we know that:

$$\Gamma; \Delta \hookrightarrow \Gamma; (!\Delta_{11}, !\Delta_{21}), (!\Delta_{12}, !\Delta_{11}, !\Delta_{22}),$$

which allows us to conclude.

□

Lemma 3.2.33 (Inversion for Sum Constructors)**.** *The following statements hold:*

1. *Let* $h \in \{\mathsf{inl}, \mathsf{inr}\}$*. If* $\Gamma; \Delta \vdash h\ M : V$*, then there exist* Δ_1, Δ_2, T, U *such that* $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$ *and* $\Gamma; \Delta_1 \vdash h\ M : T + U$ *(by a top-level application of* Val H*) and* $\Gamma; \Delta_2 \vdash T + U <: \psi(V)$*.*
2. *If* $\Gamma; \Delta \vdash T+U <: T'+U'$*, then there exist* Δ_1, Δ_2 *such that* $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2$ *and* $\Gamma; !\Delta_1 \vdash T <: T'$ *and* $\Gamma; !\Delta_2 \vdash U <: U'$*.*
3. *If* $\Gamma; \Delta \vdash \mathsf{inl}\ M : T + U$*, then there exist* $!\Delta$ *such that* $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta$ *and* $\Gamma; !\Delta \vdash M : T$ *and* $\Gamma; !\Delta \vdash U$*.*
4. *If* $\Gamma; \Delta \vdash \mathsf{inr}\ M : T + U$*, then there exist* Δ' *such that* $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$ *and* $\Gamma; !\Delta' \vdash M : U$ *and* $\Gamma; !\Delta' \vdash T$*.*
5. *If* $\Gamma; \Delta \vdash \mathsf{inl}\ M : T + U$*, then* $\Gamma; \Delta \vdash M : T$*.*

6. *If* $\Gamma; \Delta \vdash \mathsf{inr}\ M : T + U$*, then* $\Gamma; \Delta \vdash M : U$*.*

Proof. We show the six statements separately, using the first results in the proof of the later ones.

1. By induction on the derivation of $\Gamma; \Delta \vdash h\ M : V$. The proof is analogous to that of Lemma 3.2.31, part (1).

2. By induction on the derivation of $\Gamma; \Delta \vdash T + U <: T' + U'$. The proof is analogous to that of Lemma 3.2.31, part (2).

3. We know that $\Gamma; \Delta \vdash \mathsf{inl}\ M : T + U$ and that $\psi(T + U) = T + U$. We apply part (1) and derive that there exist $\Delta_1, \Delta_2, T', U'$ such that:

   - $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$,
   - $\Gamma; \Delta_1 \vdash \mathsf{inl}\ M : T' + U'$ by a top-level application of (Val Inl), and
   - $\Gamma; \Delta_2 \vdash T' + U' <: T + U$.

   By the definition of (Val Inl) the second statement lets us derive that:

   $$\Gamma; !\Delta_1' \vdash M : T',$$

   and:

   $$\Gamma; !\Delta_1' \vdash U',$$

   for some Δ_1' such that $\Gamma; \Delta_1 \hookrightarrow \Gamma; !\Delta_1'$.

   Applying part (2) to the third statement yields that there exist Δ_{21}, Δ_{22} such that:

   - $\Gamma; \Delta_2 \hookrightarrow \Gamma; !\Delta_{21}, !\Delta_{22}$,
   - $\Gamma; !\Delta_{21} \vdash T' <: T$, and
   - $\Gamma; !\Delta_{22} \vdash U' <: U$.

   We apply (Exp Subsum) to $\Gamma; !\Delta_1' \vdash M : T'$ and $\Gamma; \Delta_{21} \vdash T' <: T$, which leads to:

   $$\Gamma; !\Delta_1', !\Delta_{21} \vdash M : T.$$

   Furthermore, by Lemma 3.2.5 we know that:

   $$\Gamma; !\Delta_{22} \vdash U.$$

   Using Lemma 3.2.7 we can derive that:

   $$\Gamma; !\Delta_1', !\Delta_{21}, !\Delta_{22} \vdash M : T,$$

   and:

   $$\Gamma; !\Delta_1', !\Delta_{21}, !\Delta_{22} \vdash U.$$

   Since $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1', !\Delta_{21}, !\Delta_{22}$ by Lemma 3.2.8, we can conclude.

4. The proof follows analogously to that of statement (3).

5. The statement follows immediately by an application of statement (3) and Lemma 3.2.9.

6. The statement follows immediately by an application of statement (4) and Lemma 3.2.9.

□

Lemma 3.2.34 (Inversion for Recursive Constructors). *The following statements hold:*

1. *If* $\Gamma; \Delta \vdash \mathsf{fold}\ M : V$, *then there exist* Δ_1, Δ_2, T *such that* $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$ *and* $\Gamma; \Delta_1 \vdash \mathsf{fold}\ M : \mu\alpha. T$ *(by a top-level application of* Val Fold*) and* $\Gamma; \Delta_2 \vdash \mu\alpha. T <: \psi(V)$.

2. *If* $\Gamma; \Delta \vdash \mu\alpha. T <: \mu\alpha. T'$, *then there exists* Δ' *such that* $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$ *and* $\Gamma; !\Delta' \vdash T\{\mu\alpha. T/\alpha\} <: T'\{\mu\alpha. T'/\alpha\}$.

3. *If* $\Gamma; \Delta \vdash \mathsf{fold}\ M : \mu\alpha. T$, *then there exist* Δ' *such that* $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$ *and* $\Gamma; !\Delta' \vdash M : T\{\mu\alpha. T/\alpha\}$.

4. *If* $\Gamma; \Delta \vdash \mathsf{fold}\ M : \mu\alpha. T$, *then* $\Gamma; \Delta \vdash M : T\{\mu\alpha. T/\alpha\}$.

Proof. We show the four statements separately, using the first two results in the proof of the third.

1. By induction on the derivation of $\Gamma; \Delta \vdash \mathsf{fold}\ M : V$. The proof is analogous to that of Lemma 3.2.31, part (1).

2. By induction on the derivation of $\Gamma; \Delta \vdash \mu\alpha. T <: \mu\alpha. T'$. We implicitly use Lemma 3.2.5 whenever needed. We distinguish three cases, depending on the last applied subtyping rule:

Case (Sub Refl): In this case we know that $T = T'$ and thus we have $T\{\mu\alpha. T/\alpha\} = T'\{\mu\alpha. T'/\alpha\}$. By an application of (Sub Refl) we have $\Gamma; \emptyset \vdash T\{\mu\alpha. T/\alpha\} <: T'\{\mu\alpha. T'/\alpha\}$. Since $\Gamma; \Delta \hookrightarrow \Gamma; \emptyset$ by Lemma 3.2.8, we can conclude.

Case (Sub Pos Rec): By the premises of the subtyping rule we know that:

$$\Gamma, \alpha; !\Delta' \vdash T <: T'$$

for some Δ' such that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$. Moreover, we know that α occurs only positively in T, T'. Since $\Gamma; !\Delta' \hookrightarrow \Gamma; !\Delta'$ by Lemma 3.2.8, we can apply (Sub Pos Rec) to derive that:

$$\Gamma; !\Delta' \vdash \mu\alpha. T <: \mu\alpha. T'.$$

By point (6) of Lemma 3.2.30, we then get:

$$\Gamma; !\Delta', !\Delta' \vdash T\{\mu\alpha.T/\alpha\} <: T'\{\mu\alpha.T'/\alpha\}.$$

Since $\Gamma;\Delta \hookrightarrow \Gamma; !\Delta', !\Delta'$ by Lemma 3.2.8, we can conclude.

Case (Sub Pub Tnt): In this case we know that $\Gamma;\Delta \hookrightarrow \Gamma;\Delta_1,\Delta_2$ for some Δ_1,Δ_2 such that $\Gamma;\Delta_1 \vdash \mu\alpha.T :: \mathsf{pub}$ and $\Gamma;\Delta_2 \vdash \mu\alpha.T' :: \mathsf{tnt}$.

By the only applicable kinding rule (Kind Rec) it follows that there exist Δ'_1,Δ'_2 such that $\Gamma;\Delta_1 \hookrightarrow \Gamma;!\Delta'_1$ and $\Gamma;\Delta_2 \hookrightarrow \Gamma;!\Delta'_2$ with $\Gamma,\alpha :: \mathsf{pub};!\Delta'_1 \vdash T :: \mathsf{pub}$ and $\Gamma,\alpha :: \mathsf{tnt};!\Delta'_2 \vdash T' :: \mathsf{tnt}$.

Since $\Gamma;!\Delta'_1 \hookrightarrow \Gamma;!\Delta'_1$ and $\Gamma;!\Delta'_2 \hookrightarrow \Gamma;!\Delta'_2$ by Lemma 3.2.8, we can apply (Kind Rec) to derive that $\Gamma;!\Delta'_1 \vdash \mu\alpha.T :: \mathsf{pub}$ and $\Gamma;!\Delta'_2 \vdash \mu\alpha.T' :: \mathsf{tnt}$.

By part (4) of Lemma 3.2.30 we then get $\Gamma;!\Delta'_1,!\Delta'_1 \vdash T\{\mu\alpha.T/\alpha\} :: \mathsf{pub}$ and $\Gamma;!\Delta'_2!,!\Delta'_2 \vdash T'\{\mu\alpha.T'/\alpha\} :: \mathsf{tnt}$, hence we can apply (Sub Pub Tnt) to get:

$$\Gamma;!\Delta'_1,!\Delta'_1,!\Delta'_2,!\Delta'_2 \vdash T\{\mu\alpha.T/\alpha\} <: T'\{\mu\alpha.T'/\alpha\}.$$

Since $\Gamma;\Delta \hookrightarrow \Gamma;!\Delta'_1,!\Delta'_1,!\Delta'_2,!\Delta'_2$ by Lemma 3.2.8, we can conclude.

3. We know that $\Gamma;\Delta \vdash \mathsf{fold}\ M : \mu\alpha.T$ and that $\psi(\mu\alpha.T) = \mu\alpha.T$. We apply part (1) and derive that there exist Δ_1,Δ_2,T' such that:

   - $\Gamma;\Delta \hookrightarrow \Gamma;\Delta_1,\Delta_2$,
   - $\Gamma;\Delta_1 \vdash \mathsf{fold}\ M : \mu\alpha.T'$ by a top-level application of (Val Fold), and
   - $\Gamma;\Delta_2 \vdash \mu\alpha.T' <: \mu\alpha.T$.

   By the definition of (Val Fold) the second statement lets us derive that:

   $$\Gamma;!\Delta'_1 \vdash M : T'\{\mu\alpha.T'/\alpha\},$$

   for some Δ'_1 such that $\Gamma;\Delta_1 \hookrightarrow \Gamma;!\Delta'_1$.

   Applying part (2) to the third statement yields that there exists Δ'_2 such that:

   - $\Gamma;\Delta_2 \hookrightarrow \Gamma;!\Delta'_2$,
   - $\Gamma;!\Delta'_2 \vdash T'\{\mu\alpha.T'/\alpha\} <: T\{\mu\alpha.T/\alpha\}$.

   We apply (Exp Subsum) to $\Gamma;!\Delta'_1 \vdash M : T'\{\mu\alpha.T'/\alpha\}$ and $\Gamma;!\Delta'_2 \vdash T'\{\mu\alpha.T'/\alpha\} <: T\{\mu\alpha.T/\alpha\}$, which leads to:

   $$\Gamma;!\Delta'_1,!\Delta'_2 \vdash M : T\{\mu\alpha.T/\alpha\},$$

   Since $\Gamma;\Delta \hookrightarrow \Gamma;!\Delta'_1,!\Delta'_2$ by Lemma 3.2.8, we can conclude.

4. We can immediately conclude by an application of statement (3) and Lemma 3.2.9.

$\square$

3.2.6 Proof of subject reduction

In the following we often write $E \rightsquigarrow [\Delta \mid D]$ whenever $E \rightsquigarrow^{\widetilde{a}} [\Delta \mid D]$ for some immaterial $\widetilde{a}$ clear from the context.

Lemma 3.2.35 (Extraction and Free Values)**.** *If $E \rightsquigarrow^{\widetilde{a}} [\Delta \mid D]$, then $\mathit{fnfv}(\Delta) \cup \mathit{fnfv}(D) \subseteq \mathit{fnfv}(E)$.*

Proof. By induction on the derivation of $E \rightsquigarrow [\Delta \mid D]$. □

Lemma 3.2.36 (Extending Extraction)**.** *If $E \rightsquigarrow^{\widetilde{b}} [\Delta \mid D]$ and $a \notin \mathit{fn}(E)$, then $E \rightsquigarrow^{a,\widetilde{b}} [\Delta \mid D]$.*

Proof. By induction on the derivation of $E \rightsquigarrow^{\widetilde{b}} [\Delta \mid D]$. □

Lemma 3.2.37 (Heating Preserves Logic)**.** *If $E \Rrightarrow E'$ and $E \rightsquigarrow^{\widetilde{a}} [\Delta \mid D]$, then $E' \rightsquigarrow^{\widetilde{a}} [\Delta \mid D']$ for some D' such that $D \Rrightarrow D'$. Moreover, the depth of the derivation of $D \Rrightarrow D'$ equals that of $E \Rrightarrow E'$.*

Proof. By induction on the derivation of $E \Rrightarrow E'$:

Case (Heat Refl): the case is trivial.

Case (Heat Trans): assume $E \Rrightarrow E''$ by the premises $E \Rrightarrow E'$ and $E' \Rrightarrow E''$. Assume further $E \rightsquigarrow [\Delta \mid D]$. We apply the induction hypothesis on $E \Rrightarrow E'$ and we get $E' \rightsquigarrow [\Delta \mid D']$ with $D \Rrightarrow D'$. We then apply the induction hypothesis on $E' \Rrightarrow E''$ and we get $E'' \rightsquigarrow [\Delta \mid D'']$ with $D' \Rrightarrow D''$. Since $D \Rrightarrow D''$ by (Heat Trans), we can conclude.

Case (Heat Let): assume $\mathsf{let}\ x = E\ \mathsf{in}\ E'' \Rrightarrow \mathsf{let}\ x = E'\ \mathsf{in}\ E''$ by the premise $E \Rrightarrow E'$. Assume further $\mathsf{let}\ x = E\ \mathsf{in}\ E'' \rightsquigarrow [\Delta \mid \mathsf{let}\ x = D\ \mathsf{in}\ E'']$, which must be derived by the premise $E \rightsquigarrow [\Delta \mid D]$. We apply the induction hypothesis and we get $E' \rightsquigarrow [\Delta \mid D']$ with $D \Rrightarrow D'$. Hence, we have $\mathsf{let}\ x = E'\ \mathsf{in}\ E'' \rightsquigarrow [\Delta \mid \mathsf{let}\ x = D'\ \mathsf{in}\ E'']$ by (Extr Let) and the conclusion follows by observing that $\mathsf{let}\ x = D\ \mathsf{in}\ E'' \Rrightarrow \mathsf{let}\ x = D'\ \mathsf{in}\ E''$ by (Heat Let).

Case (Heat Res): assume $(\nu a)E \Rrightarrow (\nu a)E'$ by the premise $E \Rrightarrow E'$. Assume further $(\nu a)E \rightsquigarrow^{\widetilde{b}} [\Delta \mid (\nu a)D]$, which must be derived by the premise $E \rightsquigarrow^{a,\widetilde{b}} [\Delta \mid D]$. We apply the induction hypothesis and we get $E' \rightsquigarrow^{a,\widetilde{b}} [\Delta \mid D']$ with $D \Rrightarrow D'$. Hence, we have $(\nu a)E' \rightsquigarrow^{\widetilde{b}} [\Delta \mid (\nu a)D']$ by (Extr Res) and the conclusion follows by observing that $(\nu a)D \Rrightarrow (\nu a)D'$ by (Heat Res).

Case (Heat Fork 1): assume $E \Rsh E'' \Rrightarrow E' \Rsh E''$ by the premise $E \Rrightarrow E'$. Assume further $E \Rsh E'' \rightsquigarrow [\Delta, \Delta'' \mid D \Rsh D'']$, which must be derived by the premises $E \rightsquigarrow [\Delta \mid D]$ and $E'' \rightsquigarrow [\Delta'' \mid D'']$. By inductive hypothesis $E' \rightsquigarrow [\Delta \mid D']$ with $D \Rrightarrow D'$. Hence, we have $E' \Rsh E'' \rightsquigarrow [\Delta, \Delta'' \mid D' \Rsh D'']$ and the conclusion follows by observing that $D \Rsh D'' \Rrightarrow D' \Rsh D''$ by (Heat Fork 1).

Case (Heat Fork 2): the case is analogous to (Heat Fork 1).

Case (Heat Fork ()): assume $() \Rsh E \Rrightarrow E$. Let $E \rightsquigarrow [\Delta \mid D]$, we have $() \Rsh E \rightsquigarrow [\Delta \mid () \Rsh D]$. Since $() \Rsh D \Rrightarrow D$ by (Heat Fork ()), we can conclude. The other direction is analogous.

Case (Heat Msg ()): assume $a!M \Rrightarrow a!M \Rsh ()$. We have $a!M \rightsquigarrow [\emptyset \mid a!M]$ and $a!M \Rsh () \rightsquigarrow [\emptyset \mid a!M \Rsh ()]$, hence the conclusion follows by (Heat Msg ()).

Case (Heat Assume ()): let $\mathsf{assume}\ F \Rrightarrow \mathsf{assume}\ F \Rsh ()$. We have two possibilities: either $\mathsf{assume}\ F \rightsquigarrow [F \mid \mathsf{assume}\ \mathbf{1}]$ or $\mathsf{assume}\ F \rightsquigarrow [\emptyset \mid \mathsf{assume}\ F]$. In the first case we also have $\mathsf{assume}\ F \Rsh () \rightsquigarrow [F \mid \mathsf{assume}\ \mathbf{1} \Rsh ()]$, while in the second case we have $\mathsf{assume}\ F \Rsh () \rightsquigarrow [\emptyset \mid \mathsf{assume}\ F \Rsh ()]$. In both cases we can conclude by (Heat Assume ()).

Case (Heat Assert ()): let $\mathsf{assert}\ F \Rrightarrow \mathsf{assert}\ F \Rsh ()$. We have $\mathsf{assert}\ F \rightsquigarrow [\emptyset \mid \mathsf{assert}\ F]$ and $\mathsf{assert}\ F \Rsh () \rightsquigarrow [\emptyset \mid \mathsf{assert}\ F \Rsh ()]$, hence the conclusion follows by (Heat Assert ()).

Case (Heat Res Fork 1): assume $E \Rsh (\nu a)E' \Rrightarrow (\nu a)(E \Rsh E')$ with $a \notin \mathit{fn}(E)$. The only possible extraction derivation is the following:

$$\textsc{Extr Fork}\ \dfrac{E \rightsquigarrow^{\widetilde{b}} [\Delta \mid D] \qquad \dfrac{E' \rightsquigarrow^{a,\widetilde{b}} [\Delta' \mid D']}{(\nu a)E' \rightsquigarrow^{\widetilde{b}} [\Delta' \mid (\nu a)D']}\ \textsc{Extr Res}}{E \Rsh (\nu a)E' \rightsquigarrow^{\widetilde{b}} [\Delta, \Delta' \mid D \Rsh (\nu a)D']}$$

Since $a \notin \mathit{fn}(E)$, we can apply Lemma 3.2.36 and get $E \rightsquigarrow^{a,\widetilde{b}} [\Delta \mid D]$. Hence, we can construct the following derivation:

$$\dfrac{\textsc{Extr Fork}\ \dfrac{E \rightsquigarrow^{a,\widetilde{b}} [\Delta \mid D] \qquad E' \rightsquigarrow^{a,\widetilde{b}} [\Delta' \mid D']}{E \Rsh E' \rightsquigarrow^{a,\widetilde{b}} [\Delta, \Delta' \mid D \Rsh D']}}{(\nu a)(E \Rsh E') \rightsquigarrow^{\widetilde{b}} [\Delta, \Delta' \mid (\nu a)(D \Rsh D')]}\ \textsc{Extr Res}$$

Since $a \notin \mathit{fn}(E)$ implies $a \notin \mathit{fn}(D)$ by Lemma 3.2.35, we have $D \Rsh (\nu a)D' \Rrightarrow (\nu a)(D \Rsh D')$ by (Heat Res Fork 1) and we conclude.

Case (Heat Res Fork 2): the case is analogous to (Heat Res Fork 1).

Case (Heat Res Let): assume $\mathsf{let}\ x = (\nu a)E\ \mathsf{in}\ E' \Rrightarrow (\nu a)(\mathsf{let}\ x = E\ \mathsf{in}\ E')$ with $a \notin \mathit{fn}(E')$. The only possible extraction derivation is the following:

$$\textsc{Extr Let}\ \dfrac{\dfrac{E \rightsquigarrow^{a,\widetilde{b}} [\Delta \mid D]}{(\nu a)E \rightsquigarrow^{\widetilde{b}} [\Delta \mid (\nu a)D]}\ \textsc{Extr Res}}{\mathsf{let}\ x = (\nu a)E\ \mathsf{in}\ E' \rightsquigarrow^{\widetilde{b}} [\Delta \mid \mathsf{let}\ x = (\nu a)D\ \mathsf{in}\ E']}$$

Hence, we can construct the following derivation:

$$\dfrac{\textsc{Extr Let}\ \dfrac{E \leadsto^{a,\widetilde{b}} [\Delta \mid D]}{\mathsf{let}\ x = E\ \mathsf{in}\ E' \leadsto^{a,\widetilde{b}} [\Delta \mid \mathsf{let}\ x = D\ \mathsf{in}\ E']}}{(\nu a)(\mathsf{let}\ x = E\ \mathsf{in}\ E') \leadsto^{\widetilde{b}} [\Delta \mid (\nu a)(\mathsf{let}\ x = D\ \mathsf{in}\ E')]}\ \textsc{Extr Res}$$

Since $a \notin fn(E')$ implies $a \notin fn(D)$ by Lemma 3.2.35, we have $\mathsf{let}\ x = (\nu a)D\ \mathsf{in}\ E' \Rrightarrow (\nu a)(\mathsf{let}\ x = D\ \mathsf{in}\ E')$ by (Heat Res Let) and we conclude.

Case (Heat Fork Assoc): assume $(E \Rsh E') \Rsh E'' \Rrightarrow E \Rsh (E' \Rsh E'')$. The only possible extraction derivation is the following:

$$\textsc{Exp Fork}\ \dfrac{\textsc{Exp Fork}\ \dfrac{E \leadsto [\Delta \mid D] \qquad E' \leadsto [\Delta' \mid D']}{E \Rsh E' \leadsto [\Delta, \Delta' \mid D \Rsh D']} \qquad E'' \leadsto [\Delta'' \mid D'']}{(E \Rsh E') \Rsh E'' \leadsto [\Delta, \Delta', \Delta'' \mid (D \Rsh D') \Rsh D'']}$$

Hence, we can construct the following derivation:

$$\textsc{Exp Fork}\ \dfrac{E \leadsto [\Delta \mid D] \qquad \dfrac{E' \leadsto [\Delta' \mid D'] \qquad E'' \leadsto [\Delta'' \mid D'']}{E' \Rsh E'' \leadsto [\Delta', \Delta'' \mid D' \Rsh D'']}\ \textsc{Exp Fork}}{E \Rsh (E' \Rsh E'') \leadsto [\Delta, \Delta', \Delta'' \mid D \Rsh (D' \Rsh D'')]}$$

We observe that $(D \Rsh D') \Rsh D'' \Rrightarrow D \Rsh (D' \Rsh D'')$ by (Heat Fork Assoc) to conclude. The other direction is analogous.

Case (Heat Fork Comm): assume $(E \Rsh E') \Rsh E'' \Rrightarrow (E' \Rsh E) \Rsh E''$. The only possible extraction derivation is the following:

$$\textsc{Exp Fork}\ \dfrac{\textsc{Exp Fork}\ \dfrac{E \leadsto [\Delta \mid D] \qquad E' \leadsto [\Delta' \mid D']}{E \Rsh E' \leadsto [\Delta, \Delta' \mid D \Rsh D']} \qquad E'' \leadsto [\Delta'' \mid D'']}{(E \Rsh E') \Rsh E'' \leadsto [\Delta, \Delta', \Delta'' \mid (D \Rsh D') \Rsh D'']}$$

Hence, we can construct the following derivation:

$$\textsc{Exp Fork}\ \dfrac{\textsc{Exp Fork}\ \dfrac{E' \leadsto [\Delta' \mid D'] \qquad E \leadsto [\Delta \mid D]}{E' \Rsh E \leadsto [\Delta, \Delta' \mid D' \Rsh D]} \qquad E'' \leadsto [\Delta'' \mid D'']}{(E' \Rsh E) \Rsh E'' \leadsto [\Delta, \Delta', \Delta'' \mid (D' \Rsh D) \Rsh D'']}$$

where we note that the order of the formulas is immaterial, since we interpret the Δ's as multisets. We observe that $(D \Rsh D') \Rsh D'' \Rrightarrow (D' \Rsh D) \Rsh D''$ by (Heat Fork Comm) to conclude. The other direction is analogous.

Case (Heat Fork Let): assume $\mathsf{let}\ x = (E_1 \Rsh E_2)\ \mathsf{in}\ E_3 \Rrightarrow E_1 \Rsh (\mathsf{let}\ x = E_2\ \mathsf{in}\ E_3)$. We have $\mathsf{let}\ x = (E_1 \Rsh E_2)\ \mathsf{in}\ E_3 \leadsto [\Delta_1, \Delta_2 \mid \mathsf{let}\ x = (D_1 \Rsh D_2)\ \mathsf{in}\ E_3]$ with $E_1 \leadsto [\Delta_1 \mid D_1]$ and $E_2 \leadsto [\Delta_2 \mid D_2]$. In fact, the only possible extraction derivation is the following:

$$\text{Extr Let}\ \dfrac{\text{Extr Fork}\ \dfrac{E_1 \leadsto [\Delta_1 \mid D_1] \qquad E_2 \leadsto [\Delta_2 \mid D_2]}{E_1 \Rsh E_2 \leadsto [\Delta_1, \Delta_2 \mid D_1 \Rsh D_2]}}{\mathsf{let}\ x = (E_1 \Rsh E_2)\ \mathsf{in}\ E_3 \leadsto [\Delta_1, \Delta_2 \mid \mathsf{let}\ x = (D_1 \Rsh D_2)\ \mathsf{in}\ E_3]}$$

Hence, we can construct the following derivation:

$$\text{Extr Fork}\ \dfrac{E_1 \leadsto [\Delta_1 \mid D_1] \qquad \dfrac{E_2 \leadsto [\Delta_2 \mid D_2]}{\mathsf{let}\ x = E_2\ \mathsf{in}\ E_3 \leadsto [\Delta_2 \mid \mathsf{let}\ x = D_2\ \mathsf{in}\ E_3]}\ \text{Extr Let}}{E_1 \Rsh (\mathsf{let}\ x = E_2\ \mathsf{in}\ E_3) \leadsto [\Delta_1, \Delta_2 \mid D_1 \Rsh (\mathsf{let}\ x = D_2\ \mathsf{in}\ E_3)]}$$

Since $\mathsf{let}\ x = (D_1 \Rsh D_2)\ \mathsf{in}\ E_3 \Rrightarrow D_1 \Rsh (\mathsf{let}\ x = D_2\ \mathsf{in}\ E_3)$ by (Heat Fork Let), we can conclude. The other direction is analogous, since we can invert the construction and transform the second derivation, which is the only possible one, into the first one.

□

Lemma 3.2.38 (Total Extraction). *If $E \leadsto^{\widetilde{a}} [\Delta \mid D]$, then $D \leadsto^{\widetilde{a}} [\emptyset \mid D]$.*

Proof. By induction on the derivation of $E \leadsto^{\widetilde{a}} [\Delta \mid D]$. □

Lemma 3.2.39 (Reduction Preserves Logic). *If $E \to E'$ and $E \leadsto^{\widetilde{a}} [\Delta \mid D]$, then $D \to D'$ and $E' \leadsto^{\widetilde{a}} [\Delta, \Delta' \mid D'']$ for some D', D'', Δ' such that $D' \leadsto^{\widetilde{a}} [\Delta' \mid D^*]$ with $D^* \Rrightarrow D''$. Moreover, the depth of the derivation of $D \to D'$ equals that of $E \to E'$.*

Proof. By induction on the derivation of $E \to E'$. We note that, whenever $E \leadsto [\emptyset \mid E]$, the conclusion is trivial, hence we focus on the remaining cases:

Case (Red Let): assume $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \to \mathsf{let}\ x = E_1'\ \mathsf{in}\ E_2$ with $E_1 \to E_1'$ and $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \leadsto [\Delta_1 \mid \mathsf{let}\ x = D_1\ \mathsf{in}\ E_2]$ with $E_1 \leadsto [\Delta_1 \mid D_1]$. By induction hypothesis $D_1 \to D_1'$ and $E_1' \leadsto [\Delta_1, \Delta_1' \mid D']$ with $D_1' \leadsto [\Delta_1' \mid D_1'']$ and $D_1'' \Rrightarrow D'$. We then have $\mathsf{let}\ x = D_1\ \mathsf{in}\ E_2 \to \mathsf{let}\ x = D_1'\ \mathsf{in}\ E_2$ by (Red Let). Now we observe that $\mathsf{let}\ x = E_1'\ \mathsf{in}\ E_2 \leadsto [\Delta_1, \Delta_1' \mid \mathsf{let}\ x = D'\ \mathsf{in}\ E_2]$ and $\mathsf{let}\ x = D_1'\ \mathsf{in}\ E_2 \leadsto [\Delta_1' \mid \mathsf{let}\ x = D_1''\ \mathsf{in}\ E_2]$, so we conclude by (Heat Let).

Case (Red Res): assume $(\nu a)E \to (\nu a)E'$ with $E \to E'$ and $(\nu a)E \leadsto^{\widetilde{b}} [\Delta_1 \mid (\nu a)D_1]$ with $E \leadsto^{a,\widetilde{b}} [\Delta_1 \mid D_1]$. By induction hypothesis $D_1 \to D_1'$ and $E' \leadsto^{a,\widetilde{b}} [\Delta_1, \Delta_1' \mid D']$ with $D_1' \leadsto^{a,\widetilde{b}} [\Delta_1' \mid D_1'']$ and $D_1'' \Rrightarrow D'$. We then have $(\nu a)D_1 \to (\nu a)D_1'$ by (Red Res). Now we observe that $(\nu a)E' \leadsto^{\widetilde{b}} [\Delta_1, \Delta_1' \mid (\nu a)D']$ and $(\nu a)D_1' \leadsto^{\widetilde{b}} [\Delta_1' \mid (\nu a)D_1'']$, so we conclude by (Heat Res).

Case (RED FORK 1): assume $E_1 \upharpoonright E_2 \to E_1' \upharpoonright E_2$ with $E_1 \to E_1'$ and $E_1 \upharpoonright E_2 \rightsquigarrow [\Delta_1, \Delta_2 \mid D_1 \upharpoonright D_2]$ with $E_1 \rightsquigarrow [\Delta_1 \mid D_1]$, $E_2 \rightsquigarrow [\Delta_2 \mid D_2]$. By induction hypothesis $D_1 \to D_1'$ and $E_1' \rightsquigarrow [\Delta_1, \Delta_1' \mid D']$ with $D_1' \rightsquigarrow [\Delta_1' \mid D_1'']$ and $D_1'' \Rrightarrow D'$. We then have $D_1 \upharpoonright D_2 \to D_1' \upharpoonright D_2$ by (RED FORK 1). Now we observe that $E_1' \upharpoonright E_2 \rightsquigarrow [\Delta_1, \Delta_1', \Delta_2 \mid D' \upharpoonright D_2]$ and $D_1' \upharpoonright D_2 \rightsquigarrow [\Delta_1' \mid D_1'' \upharpoonright D_2]$, since $D_2 \rightsquigarrow [\emptyset \mid D_2]$ by Lemma 3.2.38. Thus, we conclude by (HEAT FORK 1).

Case (RED FORK 2): analogous to the previous case.

Case (RED HEAT): assume $E \to E'$ by the premises $E \Rrightarrow E_A$, $E_A \to E_B$, $E_B \Rrightarrow E'$. Assume further $E \rightsquigarrow [\Delta_1 \mid E_1]$. By Lemma 3.2.37 we have $E_A \rightsquigarrow [\Delta_1 \mid E_A']$ with $E_1 \Rrightarrow E_A'$. By inductive hypothesis we get $E_A' \to E_B'$ and $E_B \rightsquigarrow [\Delta_1, \Delta_1' \mid D_B]$ with $E_B' \rightsquigarrow [\Delta_1' \mid E_B'']$ and $E_B'' \Rrightarrow D_B$. Again by Lemma 3.2.37 we have $E' \rightsquigarrow [\Delta_1, \Delta_1' \mid E'']$ with $D_B \Rrightarrow E''$. Since we can derive $E_1 \to E_B'$ by (RED HEAT) and $E_B'' \Rrightarrow E''$ by (HEAT TRANS), we can conclude.

□

In the proof of Lemmas 3.2.42, 3.2.44, 3.2.45 and Theorem 3.2.47 below we rely on an observation about the structure of the type derivations to simplify the formal reasoning. First, we consider an alternative formulation of typing for values, presented in Table 3.1, which removes the non-structural rule (VAL REFINE). We also keep the original rules for expressions.

We can show that the original and the alternative formulation coincide.

Lemma 3.2.40 (Alternative Typing). $\Gamma; \Delta \vdash E : T$ *if and only if* $\Gamma; \Delta \vdash^{\mathsf{alt}} E : T$.

Proof. We show both directions independently:

($\Rightarrow$) By induction on the derivation of $\Gamma; \Delta \vdash E : T$:

Case (VAL VAR): let $\Gamma; \Delta \vdash x : T$ by the premises $\Gamma; \Delta \vdash \diamond$ and $(x : T) \in \Gamma$. We can construct the following type derivation:

$$\dfrac{\text{VAL VAR REFINE}\;\dfrac{(x : T) \in \Gamma \qquad \Gamma; \Delta \vdash \mathbf{1}}{\Gamma; \Delta \vdash^{\mathsf{alt}} x : \{y : T \mid \mathbf{1}\}} \qquad \dfrac{\Gamma; \emptyset \vdash \psi(T) <: \psi(T) \qquad \Gamma, y : \psi(T); \mathbf{1} \vdash \mathbf{1}}{\Gamma; \emptyset \vdash \{y : T \mid \mathbf{1}\} <: T}\;\text{SUB REFINE}}{\Gamma; \Delta \vdash^{\mathsf{alt}} x : T}\;\text{EXP SUBSUM}$$

Case (VAL REFINE): let $\Gamma; \Delta \vdash M : \{x : T \mid F\}$ by the premises $\Gamma; \Delta_1 \vdash M : T$ and $\Gamma; \Delta_2 \vdash F\{M/x\}$. By inductive hypothesis $\Gamma; \Delta_1 \vdash^{\mathsf{alt}} M : T$. By inspection of the alternative typing rules, this judgement can be derived only though an application of a structural rule after an arbitrary number of applications of (EXP SUBSUM), hence in the type derivation there must be an instance of one of the alternative type rules $\mathcal{R}$ of the form:

$$\mathcal{R}\;\dfrac{(\ldots) \qquad \Gamma; \Delta^* \vdash F'\{M/x\} \qquad \Gamma; \Delta' \hookrightarrow \Gamma; (\ldots), \Delta^*}{\Gamma; \Delta' \vdash^{\mathsf{alt}} M : \{x : U \mid F'\}}$$

$$\textsc{Val Var Refine}\quad \frac{(x : T) \in \Gamma \qquad \Gamma; \Delta \vdash F\{x/y\}}{\Gamma; \Delta \vdash^{\mathsf{alt}} x : \{y : T \mid F\}}$$

$$\textsc{Val Unit Refine}\quad \frac{\Gamma; \Delta \vdash F\{()/y\}}{\Gamma; \Delta \vdash^{\mathsf{alt}} () : \{y : \mathsf{unit} \mid F\}}$$

$$\textsc{Val Fun Refine}\quad \frac{\begin{array}{c}(\Gamma; !\Delta_1) \bullet x : T \vdash^{\mathsf{alt}} E : U\\ \Gamma; \Delta_2 \vdash F\{\lambda x.\, E/y\}\\ \Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, \Delta_2\end{array}}{\Gamma; \Delta \vdash^{\mathsf{alt}} \lambda x.\, E : \{y : x : T \to U \mid F\}}$$

$$\textsc{Val Pair Refine}\quad \frac{\begin{array}{c}\Gamma; !\Delta_1 \vdash^{\mathsf{alt}} M : T\\ \Gamma; !\Delta_2 \vdash^{\mathsf{alt}} N : U\{M/x\}\\ \Gamma; \Delta_3 \vdash F\{(M, N)/y\}\\ \Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2, \Delta_3\end{array}}{\Gamma; \Delta \vdash^{\mathsf{alt}} (M, N) : \{y : x : T * U \mid F\}}$$

$$\textsc{Val Inl Refine}\quad \frac{\begin{array}{c}\Gamma; !\Delta_1 \vdash^{\mathsf{alt}} M : T \qquad \Gamma; !\Delta_1 \vdash U\\ \Gamma; \Delta_2 \vdash F\{\mathsf{inl}\ M/y\} \qquad \Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, \Delta_2\end{array}}{\Gamma; \Delta \vdash^{\mathsf{alt}} \mathsf{inl}\ M : \{y : T + U \mid F\}}$$

$$\textsc{Val Inr Refine}\quad \frac{\begin{array}{c}\Gamma; !\Delta_1 \vdash^{\mathsf{alt}} M : U \qquad \Gamma; !\Delta_1 \vdash T\\ \Gamma; \Delta_2 \vdash F\{\mathsf{inr}\ M/y\} \qquad \Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, \Delta_2\end{array}}{\Gamma; \Delta \vdash^{\mathsf{alt}} \mathsf{inr}\ M : \{y : T + U \mid F\}}$$

$$\textsc{Val Fold Refine}\quad \frac{\begin{array}{c}\Gamma; !\Delta_1 \vdash^{\mathsf{alt}} M : T\{\mu a.\, T/\alpha\}\\ \Gamma; \Delta_2 \vdash F\{\mathsf{fold}\ M/y\} \qquad \Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, \Delta_2\end{array}}{\Gamma; \Delta \vdash^{\mathsf{alt}} \mathsf{fold}\ M : \{y : \mu\alpha.\, T \mid F\}}$$

Table 3.1: Alternative rules for typing values

where $\Gamma; \Delta_1 \hookrightarrow \Gamma; \Delta', \Delta''$ and $\Gamma; \Delta'' \vdash \{x : U \mid F'\} <: T$. (Notice that in this process we appeal to the transitivity of both the subtyping relation, proved in Lemma 3.2.21, and the environment rewriting relation, proved in Lemma 3.2.8.) Since $\Gamma; \Delta^* \vdash F'\{M/x\}$ and $\Gamma; \Delta_2 \vdash F\{M/x\}$, we know that $\Gamma; \Delta^*, \Delta_2 \vdash (F' \otimes F)\{M/x\}$ by ($\otimes$-Right), so we have:

$$\mathcal{R}\ \frac{(\ldots) \qquad \Gamma; \Delta^*, \Delta_2 \vdash (F' \otimes F)\{M/x\} \qquad \Gamma; \Delta', \Delta_2 \hookrightarrow \Gamma; (\ldots), \Delta^*, \Delta_2}{\Gamma; \Delta', \Delta_2 \vdash^{\mathsf{alt}} M : \{x : U \mid F' \otimes F\}}$$

Now we note that $\Gamma; \Delta'' \vdash \{x : U \mid F'\} <: T$ implies $\Gamma; \Delta'' \vdash \psi(U) <: \psi(T)$ by Lemma 3.2.15 in combination with Lemma 3.2.9. Hence, we also have:

$$\textsc{Sub Refine}\ \frac{\Gamma; \Delta'' \vdash \psi(U) <: \psi(T) \qquad \Gamma, x : \psi(U); F' \otimes F \vdash F}{\Gamma; \Delta'' \vdash \{x : U \mid F' \otimes F\} <: \{x : T \mid F\}}$$

Hence, $\Gamma; \Delta', \Delta_2, \Delta'' \vdash^{\mathsf{alt}} M : \{x : T \mid F\}$ by (Exp Subsum). Since we have $\Gamma; \Delta \hookrightarrow \Gamma; \Delta', \Delta'', \Delta_2$ by Lemma 3.2.8, we conclude $\Gamma; \Delta \vdash^{\mathsf{alt}} M : \{x : T \mid F\}$ by a variant of Lemma 3.2.9 predicating over the alternative typing relation.

For all the other rules for values, the proof strategy is similar to the case of (Val Var). The cases for expressions which are not values are immediate, since the two formulations share the same rules.

($\Leftarrow$) By induction on the derivation of $\Gamma; \Delta \vdash^{\mathsf{alt}} E : T$:

Case (Val Var Refine): let $\Gamma; \Delta \vdash^{\mathsf{alt}} x : \{y : T \mid F\}$ by the premises $(x : T) \in \Gamma$ and $\Gamma; \Delta \vdash F\{x/y\}$. The latter implies $\Gamma; \Delta \vdash \diamond$ by Lemma 3.2.5, hence $\Gamma; \emptyset \vdash \diamond$ again by Lemma 3.2.5 and we can conclude as follows:

$$\textsc{Val Refine}\ \dfrac{\textsc{Val Var}\ \dfrac{\Gamma; \emptyset \vdash \diamond \qquad (x : T) \in \Gamma}{\Gamma; \emptyset \vdash x : T} \qquad \Gamma; \Delta \vdash F\{x/y\}}{\Gamma; \Delta \vdash x : \{y : T \mid F\}}$$

Case (Val Fun Refine): let $\Gamma; \Delta \vdash^{\mathsf{alt}} \lambda x.\, E : \{y : x : T \to U \mid F\}$ by the premises $(\Gamma; !\Delta_1) \bullet x : T \vdash^{\mathsf{alt}} E : U$ and $\Gamma; \Delta_2 \vdash F\{\lambda x.\, E/y\}$ with $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, \Delta_2$. By inductive hypothesis $(\Gamma; !\Delta_1) \bullet x : T \vdash E : U$, hence we can conclude as follows:

$$\textsc{Val Refine}\ \dfrac{\textsc{Val Fun}\ \dfrac{(\Gamma; !\Delta_1) \bullet x : T \vdash E : U \qquad \Gamma; !\Delta_1 \hookrightarrow \Gamma; !\Delta_1}{\Gamma; !\Delta_1 \vdash \lambda x.\, E : x : T \to U} \qquad \Gamma; \Delta_2 \vdash F\{\lambda x.\, E/y\}}{\Gamma; \Delta \vdash \lambda x.\, E : \{y : x : T \to U \mid F\}}$$

The case for (Val Unit Refine) is similar to the case for (Val Var Refine). For all the other rules for values, the proof strategy is similar to the case of (Val Fun Refine). The cases for expressions which are not values are immediate, since the two formulations share the same rules.

$\square$

Now the idea is to appeal to the transitivity of both the subtyping relation (Lemma 3.2.21) and the environment rewriting relation (Lemma 3.2.8) to rearrange the structure of any type derivation constructed under the alternative typing rules. Namely, we observe that for any expression E the general form of such a type derivation is as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma; \Delta_1 \vdash^{\mathsf{alt}} E : T_1 \qquad \Gamma; \Delta_2 \vdash T_1 <: T_2 \qquad \Gamma; \Delta_3 \hookrightarrow \Gamma; \Delta_1, \Delta_2}{\vdots}}{\Gamma; \Delta_{2n-1} \vdash^{\mathsf{alt}} E : T_{2n-1}} \qquad \Gamma; \Delta_{2n} \vdash T_{2n-1} <: T \qquad \Gamma; \Delta \hookrightarrow \Gamma; \Delta_{2n-1}, \Delta_{2n}}{\Gamma; \Delta \vdash^{\mathsf{alt}} E : T}$$

where the last rule applied to derive $\Gamma; \Delta_1 \vdash^{\mathsf{alt}} E : T_1$ is not (Exp Subsum). Without loss of generality, we reorganize the derivation as follows:

$$\frac{\Gamma; \Delta_1 \vdash^{\mathsf{alt}} E : T_1 \qquad \Gamma; \Delta^* \vdash T_1 <: T \qquad \Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta^*}{\Gamma; \Delta \vdash^{\mathsf{alt}} E : T}$$

with $\Delta^* = \Delta_2, \Delta_4, \ldots, \Delta_{2n}$. Notice that also derivations which do not use rule (Exp Subsum) can be rearranged as detailed, since the subtyping relation is reflexive. Moreover, given that original typing and alternative typing coincide by Lemma 3.2.40, we note that the previous transformation can be applied to any type derivation.

Lemma 3.2.41 (Restricting Extraction)**.** *If* $E \rightsquigarrow^{\widetilde{a}} [\Delta \mid D]$ *and* $E \rightsquigarrow^{\widetilde{b}} [\Delta' \mid D']$ *with* $\{\widetilde{b}\} \subseteq \{\widetilde{a}\}$*, then* $D \rightsquigarrow^{\widetilde{b}} [\Delta'' \mid D']$*, where* $\Delta' = \Delta, \Delta''$*.*

Proof. By induction on the structure of E:

Case $E = \mathsf{assume}\ F$ with $F \neq \mathbf{1}$ and $fn(F) \cap \{\widetilde{a}\} = \emptyset$: we have $E \rightsquigarrow^{\widetilde{a}} [F \mid \mathsf{assume}\ \mathbf{1}]$ by (Extr Assume). Since $\{\widetilde{b}\} \subseteq \{\widetilde{a}\}$, we know that $fn(F) \cap \{\widetilde{b}\} = \emptyset$, hence we have $E \rightsquigarrow^{\widetilde{b}} [F \mid \mathsf{assume}\ \mathbf{1}]$ by (Extr Assume). We know that $\mathsf{assume}\ \mathbf{1} \rightsquigarrow^{\widetilde{b}} [\emptyset \mid \mathsf{assume}\ \mathbf{1}]$ by (Extr Exp), which allows us to conclude.

Case $E = \mathsf{assume}\ F$ with $F \neq \mathbf{1}$ and $fn(F) \cap \{\widetilde{a}\} \neq \emptyset$: we have $E \rightsquigarrow^{\widetilde{a}} [\emptyset \mid \mathsf{assume}\ F]$ by (Extr Exp). Now we distinguish two cases: if $fn(F) \cap \{\widetilde{b}\} \neq \emptyset$, then we also have $E \rightsquigarrow^{\widetilde{b}} [\emptyset \mid \mathsf{assume}\ F]$ by (Extr Exp), i.e., we have $\mathsf{assume}\ F \rightsquigarrow^{\widetilde{b}} [\emptyset \mid \mathsf{assume}\ F]$ and we conclude. Otherwise, whenever $fn(F) \cap \{\widetilde{b}\} = \emptyset$, we have $E \rightsquigarrow^{\widetilde{b}} [F \mid \mathsf{assume}\ \mathbf{1}]$ by (Extr Assume), i.e., we have $\mathsf{assume}\ F \rightsquigarrow^{\widetilde{b}} [F \mid \mathsf{assume}\ \mathbf{1}]$ and we conclude again.

Case $E = E_1 \upharpoonleft E_2$: we know by the definition of the only applicable extraction rule (Extr Fork) that:

- $E \rightsquigarrow^{\widetilde{a}} [\Delta_1, \Delta_2 \mid D_1 \upharpoonleft D_2]$ and
- $E \rightsquigarrow^{\widetilde{b}} [\Delta_1', \Delta_2' \mid D_1' \upharpoonleft D_2']$, where
- $E_i \rightsquigarrow^{\widetilde{a}} [\Delta_i \mid D_i]$ and
- $E_i \rightsquigarrow^{\widetilde{b}} [\Delta_i' \mid D_i']$ for $i \in \{1, 2\}$.

By applying the induction hypothesis to the latter two statements we know that there exist Δ_1'', Δ_2'' such that:

$$D_i \rightsquigarrow^{\widetilde{b}} [\Delta_i'' \mid D_i'],$$

where $\Delta_i' = \Delta_i, \Delta_i''$ for $i \in \{1, 2\}$. By (Exp Fork) we can conclude that:

$$D_1 \upharpoonleft D_2 \rightsquigarrow^{\widetilde{b}} [\Delta_1'', \Delta_2'' \mid D_1' \upharpoonleft D_2'],$$

where $\Delta_1', \Delta_2' = \Delta_1, \Delta_1'', \Delta_2, \Delta_1'' = \Delta_1, \Delta_2, \Delta_1'', \Delta_2''$.

Case E is a restriction or let: in this case both $E \leadsto^{\widetilde{a}} [\Delta \mid D]$ and $E \leadsto^{\widetilde{b}} [\Delta' \mid D']$ must have been derived by a top-level application of the same extraction rule $\mathcal{R}$. We apply the induction hypothesis to the premise of the extraction rule $\mathcal{R}$ and conclude by applying $\mathcal{R}$ to the result, similarly to the previous case of forks.

Case E has a different form: in this case both $E \leadsto^{\widetilde{a}} [\emptyset \mid E]$ and $E \leadsto^{\widetilde{b}} [\emptyset \mid E]$ by (Extr Exp), so we immediately conclude.

□

Lemma 3.2.42 (Extraction Preserves Typing). *If* $\Gamma; \Delta \vdash E : T$ *and* $E \leadsto^{\widetilde{a}} [\Delta' \mid E']$, *then* $\Gamma; \Delta, \Delta' \vdash E' : T$.

Proof. By a case analysis on the structure of E:

Case E is any expression such that $E \leadsto [\emptyset \mid E]$: the conclusion is trivial.

Case $E = \mathsf{assume}\ F$ with $F \neq \mathbf{1}$ and $fn(F) \cap \{\widetilde{a}\} = \emptyset$: we have $E \leadsto^{\widetilde{a}} [F \mid \mathsf{assume}\ \mathbf{1}]$ and $\Gamma; \Delta \vdash \mathsf{assume}\ F : T$. The typing judgement must follow by an instance of (Exp Assume) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash \mathsf{assume}\ F : U$ and $\Gamma; \Delta_B \vdash U <: T$ with $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$ and $\Gamma; \Delta_A, F \vdash \mathsf{assume}\ \mathbf{1} : U$. The conclusion $\Gamma; \Delta, F \vdash \mathsf{assume}\ \mathbf{1} : T$ follows by (Exp Subsum).

Case $E = (\nu a)D$: we have $E \leadsto^{\widetilde{b}} [\Delta' \mid (\nu a)D']$ with $D \leadsto^{a,\widetilde{b}} [\Delta' \mid D']$ and $\Gamma; \Delta \vdash (\nu a)D : T$. The typing judgement must follow by an instance of (Exp Res) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash (\nu a)D : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $D \leadsto^{a} [\Delta'' \mid D'']$
- $\Gamma, a \updownarrow T'; \Delta_A, \Delta'' \vdash D'' : V$

By Lemma 3.2.41 we know that $D' \leadsto^{a} [\Delta''' \mid D'']$, for some Δ''' such that $\Delta'' = \Delta', \Delta'''$. We can then construct the following type derivation:

$$\dfrac{\dfrac{D' \leadsto^{a} [\Delta''' \mid D''] \qquad \Gamma, a \updownarrow T'; \Delta_A, \overbrace{\Delta', \Delta'''}^{\Delta''} \vdash D'' : V}{\Gamma; \Delta_A, \Delta' \vdash (\nu a)D' : V}\ \text{Exp Res} \qquad \Gamma; \Delta_B \vdash V <: T}{\Gamma; \Delta, \Delta' \vdash (\nu a)D' : T}\ \text{Exp Subsum}$$

Case $E = \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$: we have $E \leadsto^{\widetilde{a}} [\Delta' \mid \mathsf{let}\ x = D'\ \mathsf{in}\ E_2]$ with $E_1 \leadsto^{\widetilde{a}} [\Delta' \mid D']$ and $\Gamma; \Delta \vdash \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : T$. The typing judgement must follow by an instance of (Exp Let) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $E_1 \rightsquigarrow^{\emptyset} [\Delta'' \mid D'']$
- $\Gamma; \Delta_A, \Delta'' \hookrightarrow \Gamma; \Delta_1, \Delta_2$
- $\Gamma; \Delta_1 \vdash D'' : U$
- $(\Gamma; \Delta_2) \bullet x : U \vdash E_2 : V$

By Lemma 3.2.41 we know that $D' \rightsquigarrow^{\emptyset} [\Delta''' \mid D'']$, for some Δ''' such that $\Delta'' = \Delta', \Delta'''$. Hence, we have $\Gamma; \Delta_A, \Delta', \Delta''' \hookrightarrow \Gamma; \Delta_1, \Delta_2$ and we can then construct the following type derivation:

$$\dfrac{\textsc{Exp Let}\ \dfrac{D' \rightsquigarrow^{\emptyset} [\Delta''' \mid D''] \qquad \Gamma; \Delta_1 \vdash D'' : U \qquad (\Gamma; \Delta_2) \bullet x : U \vdash E_2 : V}{\Gamma; \Delta_A, \Delta' \vdash \mathsf{let}\ x = D'\ \mathsf{in}\ E_2 : V} \qquad \Gamma; \Delta_B \vdash V <: T}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ x = D'\ \mathsf{in}\ E_2 : T}\ \textsc{Exp Subsum}$$

Case $E = E_1 \Rsh E_2$: similar to the previous case.

□

Lemma 3.2.43 (Transitivity of Extraction). *Let* $E \rightsquigarrow^{\widetilde{b}} [\Delta' \mid E']$ *and* $E' \rightsquigarrow^{\widetilde{c}} [\Delta'' \mid E'']$, *where* $\{\widetilde{c}\} \subseteq \{\widetilde{b}\}$, *then* $E \rightsquigarrow^{\widetilde{c}} [\Delta', \Delta'' \mid E'']$.

Proof. By induction on the structure of E:

Case $E = \mathsf{assume}\ F$, where $F \neq \mathbf{1}$ and $\mathit{fn}(F) \cap \{\widetilde{b}\} = \emptyset$. In this case we know, by definition of the only applicable extraction rule (Extr Assume), that $E \rightsquigarrow^{\widetilde{b}} [\Delta' \mid E']$ with $\Delta' = F$ and $E' = \mathsf{assume}\ \mathbf{1}$. It immediately follows by the only applicable extraction rule (Extr Exp) that $E' \rightsquigarrow^{\widetilde{c}} [\Delta'' \mid E'']$ with $\Delta'' = \emptyset$ and $E'' = \mathsf{assume}\ \mathbf{1}$. Since we know that $\{\widetilde{c}\} \subseteq \{\widetilde{b}\}$ and $\mathit{fn}(F) \cap \{\widetilde{b}\} = \emptyset$, we know that $\mathit{fn}(F) \cap \{\widetilde{c}\} = \emptyset$. We can thus apply (Extr Assume) to derive $E \rightsquigarrow^{\widetilde{c}} [F \mid \mathsf{assume}\ \mathbf{1}]$ and conclude.

Case E is a restriction, fork, or let: in this case both $E \rightsquigarrow^{\widetilde{b}} [\Delta' \mid E']$ and $E' \rightsquigarrow^{\widetilde{c}} [\Delta'' \mid E'']$ must have been derived by a top-level application of the same extraction rule $\mathcal{R}$. We apply the induction hypothesis to the premise(s) of the extraction rule $\mathcal{R}$ and conclude by applying $\mathcal{R}$ to the result(s).

Case E has a different form: In this case we know that $E \rightsquigarrow^{\widetilde{b}} [\emptyset \mid E']$ with $E' = E$. Since we know that $E' \rightsquigarrow^{\widetilde{c}} [\Delta' \mid E'']$, it immediately follows that $E \rightsquigarrow^{\widetilde{c}} [\Delta' \mid E'']$ and we conclude.

□

Lemma 3.2.44 (Inverting Extraction Preserves Typing). *Let* $E \rightsquigarrow^{\widetilde{b}} [\Delta' \mid E']$. *If* $\Gamma; \Delta, \Delta' \vdash E' : T$, *then* $\Gamma; \Delta \vdash E : T$.

Proof. By a case analysis on the structure of E:

Case E is any expression such that $E \rightsquigarrow [\emptyset \mid E]$: the conclusion is trivial.

Case $E = \mathsf{assume}\ F$ with $F \neq \mathbf{1}$ and $fn(F) \cap \{\widetilde{b}\} = \emptyset$: we know that $E \rightsquigarrow^{\widetilde{b}} [F \mid \mathsf{assume}\ \mathbf{1}]$ and $\Gamma; \Delta, F \vdash \mathsf{assume}\ \mathbf{1} : T$. The conclusion $\Gamma; \Delta \vdash \mathsf{assume}\ F : T$ immediately follows by (Exp Assume).

Case $E = (\nu a)D$: we have $E \rightsquigarrow^{\widetilde{b}} [\Delta' \mid (\nu a)D']$ with $D \rightsquigarrow^{a,\widetilde{b}} [\Delta' \mid D']$ and $\Gamma; \Delta, \Delta' \vdash (\nu a)D' : T$. The typing judgement must follow by an instance of (Exp Res) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash (\nu a)D' : V$ by a top-level application of (Exp Res) and $\Gamma; \Delta_B \vdash V <: T$ with $\Gamma; \Delta, \Delta' \hookrightarrow \Gamma; \Delta_A, \Delta_B$.

Since we know that $\Gamma; \Delta_A \vdash (\nu a)D' : V$ by (Exp Res), it must be the case that $D' \rightsquigarrow^{a} [\Delta'' \mid D'']$ and $\Gamma, a \updownarrow W; \Delta_A, \Delta'' \vdash D'' : V$ with $a \notin fn(V)$.

By Lemma 3.2.43 we know that $D \rightsquigarrow^{a,\widetilde{b}} [\Delta' \mid D']$ and $D' \rightsquigarrow^{a} [\Delta'' \mid D'']$ imply:

$$D \rightsquigarrow^{a} [\Delta', \Delta'' \mid D''].$$

By Lemma 3.2.7 we know that $\Gamma; \Delta_B \vdash V <: T$ implies:

$$\Gamma, a \updownarrow W; \Delta_B \vdash V <: T.$$

Applying (Exp Subsum) to the latter and $\Gamma, a \updownarrow W; \Delta_A, \Delta'' \vdash D'' : V$, we get:

$$\Gamma, a \updownarrow W; \Delta_A, \Delta'', \Delta_B \vdash D'' : T.$$

We observe that $\Gamma, a \updownarrow W; \Delta, \Delta', \Delta'' \hookrightarrow \Gamma, a \updownarrow W; \Delta_A, \Delta'', \Delta_B$, so we can apply Lemma 3.2.9 and get:

$$\Gamma, a \updownarrow W; \Delta, \Delta', \Delta'' \vdash D'' : T.$$

Finally, we note that $a \notin fn(T)$ by applying Lemma 3.2.5 to $\Gamma; \Delta_B \vdash V <: T$, hence we conclude $\Gamma; \Delta \vdash (\nu a)D : T$ by an application of (Exp Res).

Case $E = \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$: We know that $E \rightsquigarrow^{\widetilde{b}} [\Delta' \mid \mathsf{let}\ x = D_1\ \mathsf{in}\ E_2]$, where $E_1 \rightsquigarrow^{\widetilde{b}} [\Delta' \mid D_1]$ and $\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ x = D_1\ \mathsf{in}\ E_2 : T$.

The typing judgement must follow by an instance of (Exp Let) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash \mathsf{let}\ x = D_1\ \mathsf{in}\ E_2 : V$ by a top-level application of (Exp Let) and $\Gamma; \Delta_B \vdash V <: T$ with $\Gamma; \Delta, \Delta' \hookrightarrow \Gamma; \Delta_A, \Delta_B$.

Since we know that $\Gamma; \Delta_A \vdash \mathsf{let}\ x = D_1\ \mathsf{in}\ E_2 : V$ by (Exp Let), it must be the case that $D_1 \rightsquigarrow^{\emptyset} [\Delta'' \mid D_1']$ and $\Gamma; \Delta_1 \vdash D_1' : W$ and $(\Gamma; \Delta_2) \bullet x : W \vdash E_2 : V$, for some Δ_1, Δ_2 such that $\Gamma; \Delta_A, \Delta'' \hookrightarrow \Gamma; \Delta_1, \Delta_2$.

By Lemma 3.2.43 we know that $E_1 \rightsquigarrow^{\tilde{b}} [\Delta' \mid D_1]$ and $D_1 \rightsquigarrow^{\emptyset} [\Delta'' \mid D_1']$ imply:

$$E_1 \rightsquigarrow^{\emptyset} [\Delta', \Delta'' \mid D_1'].$$

By Lemma 3.2.7 we know that $\Gamma; \Delta_B \vdash V <: T$ implies:

$$\Gamma, x : \psi(W); \Delta_B \vdash V <: T.$$

Applying (Exp Subsum) to the latter and $(\Gamma; \Delta_2) \bullet x : W \vdash E_2 : V$, we get:

$$(\Gamma; \Delta_2, \Delta_B) \bullet x : W \vdash E_2 : T.$$

We conclude $\Gamma; \Delta \vdash \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : T$ by applying (Exp Let) to the collected statements:

- $E_1 \rightsquigarrow^{\emptyset} [\Delta', \Delta'' \mid D_1']$
- $\Gamma; \Delta_1 \vdash D_1' : W$
- $(\Gamma; \Delta_2, \Delta_B) \bullet x : W \vdash E_2 : T$, and
- $\Gamma; \Delta, \Delta', \Delta'' \hookrightarrow \Gamma; \Delta_1, (\Delta_2, \Delta_B)$, which holds by Lemma 3.2.8.

Case $E = E_1 \upharpoonright E_2$: similar to the previous case.

□

Lemma 3.2.45 (Heating Preserves Typing). *If $\Gamma; \Delta \vdash E : T$ and $E \Rrightarrow E'$, then $\Gamma; \Delta \vdash E' : T$.*

Proof. By induction on the derivation of $E \Rrightarrow E'$:

Case (Heat Refl): the case is trivial.

Case (Heat Trans): assume $E \Rrightarrow E''$ by the premises $E \Rrightarrow E'$ and $E' \Rrightarrow E''$. Assume further that $\Gamma; \Delta \vdash E : T$. We apply the inductive hypothesis twice and we conclude $\Gamma; \Delta \vdash E'' : T$.

Case (Heat Let): assume $\mathsf{let}\ x = E\ \mathsf{in}\ E'' \Rrightarrow \mathsf{let}\ x = E'\ \mathsf{in}\ E''$ by the premise $E \Rrightarrow E'$. Assume further that $\Gamma; \Delta \vdash \mathsf{let}\ x = E\ \mathsf{in}\ E'' : T$, which must follow by an instance of (Exp Let) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash \mathsf{let}\ x = E\ \mathsf{in}\ E'' : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $E \rightsquigarrow [\Delta' \mid D]$
- $\Gamma; \Delta, \Delta' \hookrightarrow \Gamma; \Delta_1, \Delta_2$
- $\Gamma; \Delta_1 \vdash D : U$

- $(\Gamma; \Delta_2) \bullet x : U \vdash E'' : V$

By Lemma 3.2.37 we know that $E \Rrightarrow E'$ implies $E' \rightsquigarrow [\Delta' \mid D']$ with $D \Rrightarrow D'$. Since Lemma 3.2.37 is depth-preserving, we can apply the inductive hypothesis and get $\Gamma; \Delta_1 \vdash D' : U$, hence the conclusion $\Gamma; \Delta \vdash \mathsf{let}\ x = E'\ \mathsf{in}\ E'' : T$ follows by applying (Exp Let) and (Exp Subsum).

Case (Heat Res): assume $(\nu a)E \Rrightarrow (\nu a)E'$ by the premise $E \Rrightarrow E'$. Assume further that $\Gamma; \Delta \vdash (\nu a)E : T$. The typing judgement must follow by an instance of (Exp Res) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash (\nu a)E : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $E \rightsquigarrow^a [\Delta' \mid D]$
- $\Gamma, a \updownarrow T'; \Delta, \Delta' \vdash D : V$

By Lemma 3.2.37 we know that $E \Rrightarrow E'$ implies $E' \rightsquigarrow^a [\Delta' \mid D']$ with $D \Rrightarrow D'$. Since Lemma 3.2.37 is depth-preserving, we can apply the inductive hypothesis and get $\Gamma, a \updownarrow T'; \Delta, \Delta' \vdash D' : V$, hence the conclusion $\Gamma; \Delta \vdash (\nu a)E' : T$ follows by applying (Exp Res) and (Exp Subsum).

Case (Heat Fork 1): assume $E \Rsh E'' \Rrightarrow E' \Rsh E''$ by the premise $E \Rrightarrow E'$. Assume further that $\Gamma; \Delta \vdash E \Rsh E'' : T$. The judgement must follow by an instance of (Exp Fork) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash E \Rsh E'' : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $E \rightsquigarrow [\Delta' \mid D]$
- $E'' \rightsquigarrow [\Delta'' \mid D'']$
- $\Gamma; \Delta_A, \Delta', \Delta'' \hookrightarrow \Gamma; \Delta_1, \Delta_2$
- $\Gamma; \Delta_1 \vdash D : U$
- $\Gamma; \Delta_2 \vdash D'' : V$

By Lemma 3.2.37 we know that $E \Rrightarrow E'$ implies $E' \rightsquigarrow [\Delta' \mid D']$ with $D \Rrightarrow D'$. Since Lemma 3.2.37 is depth-preserving, we can apply the inductive hypothesis and get $\Gamma; \Delta_1 \vdash D' : U$, hence the conclusion $\Gamma; \Delta \vdash E' \Rsh E'' : T$ follows by applying (Exp Fork) and (Exp Subsum).

Case (Heat Fork 2): the case is analogous to Heat Fork 1.

Case (Heat Fork ()): assume $() \Rsh E \Rrightarrow E$ with $\Gamma; \Delta \vdash () \Rsh E : T$. The judgement must follow by an instance of (Exp Fork) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash () \Rsh E : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $() \rightsquigarrow [\emptyset \mid ()]$
- $E \rightsquigarrow [\Delta' \mid E']$
- $\Gamma; \Delta_A, \Delta' \hookrightarrow \Gamma; \Delta_1, \Delta_2$
- $\Gamma; \Delta_1 \vdash () : U$
- $\Gamma; \Delta_2 \vdash E' : V$

Notice that both $\Gamma; \Delta_1 \vdash \diamond$ and $\Gamma; \Delta_2 \vdash \diamond$ by Lemma 3.2.5, thus $\Gamma; \Delta_1, \Delta_2 \vdash \diamond$ by Lemma 3.2.6. By Lemma 3.2.7 we then know that $\Gamma; \Delta_2 \vdash E' : V$ implies $\Gamma; \Delta_1, \Delta_2 \vdash E' : V$, hence we have $\Gamma; \Delta_A, \Delta' \vdash E' : V$ by Lemma 3.2.9 and this implies $\Gamma; \Delta_A \vdash E : V$ by Lemma 3.2.44. The conclusion $\Gamma; \Delta \vdash E : T$ follows by (Exp Subsum).

Assume now $E \Rrightarrow () \upharpoonright E$ with $\Gamma; \Delta \vdash E : T$. The judgement must follow by an instance of a structural rule after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash E : V$ and $\Gamma; \Delta_B \vdash V <: T$ with $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$. By Lemma 3.2.5 we know that $\Gamma; \Delta_A \vdash E : V$ implies $\Gamma; \Delta_A \vdash \diamond$, hence $\Gamma; \emptyset \vdash \diamond$ again by Lemma 3.2.5 and $\Gamma; \emptyset \vdash () : \mathsf{unit}$ by (Val Unit). Let then $E \rightsquigarrow [\Delta' \mid E']$: since $\Gamma; \Delta_A \vdash E : V$, we have $\Gamma; \Delta_A, \Delta' \vdash E' : V$ by Lemma 3.2.42. Hence, we have:

- $() \rightsquigarrow [\emptyset \mid ()]$
- $E \rightsquigarrow [\Delta' \mid E']$
- $\Gamma; \emptyset \vdash () : \mathsf{unit}$
- $\Gamma; \Delta_A, \Delta' \vdash E' : V$

which imply $\Gamma; \Delta_A \vdash () \upharpoonright E : V$ by (Exp Fork). The conclusion $\Gamma; \Delta \vdash () \upharpoonright E : T$ follows by (Exp Subsum).

Case (Heat Msg ()): let $a!M \Rrightarrow a!M \upharpoonright ()$ with $\Gamma; \Delta \vdash a!M : T$. The judgement must follow by an instance of (Exp Send) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash a!M : \mathsf{unit}$ and $\Gamma; \Delta_B \vdash \mathsf{unit} <: T$ with $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$. By Lemma 3.2.5 we know that $\Gamma; \Delta_A \vdash a!M : \mathsf{unit}$ implies $\Gamma; \Delta_A \vdash \diamond$, hence $\Gamma; \emptyset \vdash \diamond$ again by Lemma 3.2.5 and $\Gamma; \emptyset \vdash () : \mathsf{unit}$ by (Val Unit). Thus, we have:

- $a!M \rightsquigarrow [\emptyset \mid a!M]$
- $() \rightsquigarrow [\emptyset \mid ()]$
- $\Gamma; \Delta_A \vdash a!M : \mathsf{unit}$
- $\Gamma; \emptyset \vdash () : \mathsf{unit}$

which imply $\Gamma; \Delta_A \vdash a!M \upharpoonright () : \mathsf{unit}$ by (Exp Fork). Hence, the conclusion $\Gamma; \Delta \vdash a!M \upharpoonright () : T$ follows by (Exp Subsum).

Case (Heat Assume ()): let $\mathsf{assume}\ F \Rrightarrow \mathsf{assume}\ F \Rsh ()$ with $\Gamma;\Delta \vdash \mathsf{assume}\ F : T$. We distinguish two cases. Let $F = \mathbf{1}$, then $\Gamma;\Delta \vdash \mathsf{assume}\ \mathbf{1} : T$ must follow by an instance of (Exp True) after an instance of (Exp Subsum), hence it must be the case that $\Gamma;\Delta_A \vdash \mathsf{assume}\ \mathbf{1} : \mathsf{unit}$ and $\Gamma;\Delta_B \vdash \mathsf{unit} <: T$ with $\Gamma;\Delta \hookrightarrow \Gamma;\Delta_A,\Delta_B$. Now notice that $\mathsf{assume}\ \mathbf{1} \rightsquigarrow [\emptyset \mid \mathsf{assume}\ \mathbf{1}]$ and $() \rightsquigarrow [\emptyset \mid ()]$, hence we can construct the following type derivation:

$$\dfrac{\textsc{Exp Fork}\ \dfrac{\Gamma;\Delta_A \vdash \mathsf{assume}\ \mathbf{1} : \mathsf{unit} \qquad \dfrac{\Gamma;\emptyset \vdash \diamond}{\Gamma;\emptyset \vdash () : \mathsf{unit}}\ \textsc{Val Unit}}{\Gamma;\Delta_A \vdash \mathsf{assume}\ \mathbf{1} \Rsh () : \mathsf{unit}} \qquad \Gamma;\Delta_B \vdash \mathsf{unit} <: T}{\Gamma;\Delta \vdash \mathsf{assume}\ \mathbf{1} \Rsh () : T}\ \textsc{Exp Subsum}$$

Let now $F \neq \mathbf{1}$, then $\Gamma;\Delta \vdash \mathsf{assume}\ F : T$ must follow by an instance of (Exp Assume) after an instance of (Exp Subsum), hence it must be the case that $\Gamma;\Delta_A \vdash \mathsf{assume}\ F : V$ and $\Gamma;\Delta_B \vdash V <: T$ with $\Gamma;\Delta \hookrightarrow \Gamma;\Delta_A,\Delta_B$ and $\Gamma;\Delta_A, F \vdash \mathsf{assume}\ \mathbf{1} : V$. The latter must have been derived by an instance of (Exp True) after an instance of (Exp Subsum), hence we have $\Gamma;\Delta_1 \vdash \mathsf{assume}\ \mathbf{1} : \mathsf{unit}$ and $\Gamma;\Delta_2 : \mathsf{unit} <: V$ with $\Gamma;\Delta_A, F \hookrightarrow \Gamma;\Delta_1,\Delta_2$. Now notice that $\mathsf{assume}\ F \rightsquigarrow [F \mid \mathsf{assume}\ \mathbf{1}]$ and $() \rightsquigarrow [\emptyset \mid ()]$, hence we can construct the following type derivation:

$$\dfrac{\textsc{Exp Fork}\ \dfrac{\textsc{Exp True}\ \dfrac{\Gamma;\emptyset \vdash \diamond}{\Gamma;\emptyset \vdash \mathsf{assume}\ \mathbf{1} : \mathsf{unit}} \qquad \dfrac{\textsc{Val Unit}\ \dfrac{\Gamma;\Delta_1 \vdash \diamond}{\Gamma;\Delta_1 \vdash () : \mathsf{unit}} \qquad \Gamma;\Delta_2 \vdash \mathsf{unit} <: V}{\Gamma;\Delta_A, F \vdash () : V}\ \textsc{Exp Subsum}}{\Gamma;\Delta_A \vdash \mathsf{assume}\ F \Rsh () : V} \qquad \Gamma;\Delta_B \vdash V <: T}{\Gamma;\Delta \vdash \mathsf{assume}\ F \Rsh () : T}\ \textsc{Exp Subsum}$$

Case (Heat Assert ()): the case is analogous to (Heat Msg ()).

Case (Heat Res Fork 1): let $E' \Rsh (\nu a)E \Rrightarrow (\nu a)(E' \Rsh E)$ with $a \notin \mathit{fn}(E')$. Assume further $\Gamma;\Delta \vdash E' \Rsh (\nu a)E : T$. The judgement must follow by an instance of (Exp Fork) after an instance of (Exp Subsum), hence it must be the case that $\Gamma;\Delta_A \vdash E' \Rsh (\nu a)E : V$ and $\Gamma;\Delta_B \vdash V <: T$ with:

- $\Gamma;\Delta \hookrightarrow \Gamma;\Delta_A,\Delta_B$
- $E' \rightsquigarrow^{\emptyset} [\Delta' \mid D']$
- $(\nu a)E \rightsquigarrow^{\emptyset} [\Delta'' \mid (\nu a)D]$ with $E \rightsquigarrow^{a} [\Delta'' \mid D]$
- $\Gamma;\Delta_A,\Delta',\Delta'' \hookrightarrow \Gamma;\Delta_1,\Delta_2$
- $\Gamma;\Delta_1 \vdash D' : U$
- $\Gamma;\Delta_2 \vdash (\nu a)D : V$

The latter judgement must follow by an instance of (Exp Res) after an instance of (Exp Subsum). Notice that $E \rightsquigarrow^a [\Delta'' \mid D]$ implies $D \rightsquigarrow^a [\emptyset \mid D]$ by Lemma 3.2.38, hence we simply have $\Gamma, a \updownarrow T'; \Delta_{21} \vdash D : U$ and $\Gamma; \Delta_{22} \vdash U <: V$ with $\Gamma; \Delta_2 \hookrightarrow \Gamma; \Delta_{21}, \Delta_{22}$. We also notice that $E' \rightsquigarrow^\emptyset [\Delta' \mid D']$ and $a \notin fn(E')$ imply $E' \rightsquigarrow^a [\Delta' \mid D']$ by Lemma 3.2.36, hence $E' \Rsh E \rightsquigarrow^a [\Delta', \Delta'' \mid D' \Rsh D]$ by (Extr Fork). Moreover, we know that $E' \rightsquigarrow^\emptyset [\Delta' \mid D']$ implies $D' \rightsquigarrow^\emptyset [\emptyset \mid D']$ by Lemma 3.2.38, hence we can construct the following type derivation:

$$\text{Exp Res}\ \dfrac{\text{Exp Fork}\ \dfrac{D \rightsquigarrow^\emptyset [\Delta''' \mid D''] \qquad \dfrac{\Gamma; \Delta_1 \vdash D' : U}{\Gamma, a \updownarrow T'; \Delta_1 \vdash D' : U} \qquad \dfrac{(1)}{\Gamma, a \updownarrow T'; \Delta_2, \Delta''' \vdash D'' : V}}{\Gamma, a \updownarrow T'; \Delta_A, \Delta', \Delta'' \vdash D' \Rsh D : V}}{\Gamma; \Delta_A \vdash (\nu a)(E' \Rsh E) : V}$$

where (1) is constructed as follows:

$$\text{Exp Subsum}\ \dfrac{\dfrac{(2)}{\Gamma, a \updownarrow T'; \Delta_{21}, \Delta''' \vdash D'' : U} \qquad \dfrac{\Gamma; \Delta_{22} \vdash U <: V}{\Gamma, a \updownarrow T'; \Delta_{22} \vdash U <: V}}{\Gamma, a \updownarrow T'; \Delta_2, \Delta''' \vdash D'' : V}$$

and (2) is derived from $\Gamma, a \updownarrow T'; \Delta_{21} \vdash D : U$ and $D \rightsquigarrow^\emptyset [\Delta''' \mid D'']$ using Lemma 3.2.42. We conclude $\Gamma; \Delta \vdash (\nu a)(E' \Rsh E) : T$ by (Exp Subsum).

Case (Heat Res Fork 2): the case is analogous to (Heat Res Fork 1).

Case (Heat Res Let): assume $\mathsf{let}\ x = (\nu a)E\ \mathsf{in}\ E' \Rrightarrow (\nu a)(\mathsf{let}\ x = E\ \mathsf{in}\ E')$ with $a \notin fn(E')$. Assume further $\Gamma; \Delta \vdash \mathsf{let}\ x = (\nu a)E\ \mathsf{in}\ E' : T$. The judgement must follow by an instance of (Exp Let) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash \mathsf{let}\ x = (\nu a)E\ \mathsf{in}\ E' : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $(\nu a)E \rightsquigarrow^\emptyset [\Delta' \mid (\nu a)D]$ with $E \rightsquigarrow^a [\Delta' \mid D]$
- $\Gamma; \Delta_A, \Delta' \hookrightarrow \Gamma; \Delta_1, \Delta_2$
- $\Gamma; \Delta_1 \vdash (\nu a)D : U$
- $(\Gamma; \Delta_2) \bullet x : U \vdash E' : V$

Now we note that $\Gamma; \Delta_1 \vdash (\nu a)D : U$ must follow by an instance of (Exp Res) after an instance of (Exp Subsum). Since $E \rightsquigarrow^a [\Delta' \mid D]$ implies $D \rightsquigarrow^a [\emptyset \mid D]$ by Lemma 3.2.38, we note that we simply have $\Gamma, a \updownarrow T'; \Delta_{11} \vdash D : U'$ and $\Gamma; \Delta_{12} \vdash U' <: U$ with $\Gamma; \Delta_1 \hookrightarrow \Gamma; \Delta_{11}, \Delta_{12}$. Notice also that $E \rightsquigarrow^a [\Delta' \mid D]$ implies $\mathsf{let}\ x = E\ \mathsf{in}\ E' \rightsquigarrow^a [\Delta' \mid \mathsf{let}\ x = D\ \mathsf{in}\ E']$ by (Extr Let). We can then construct the following type derivation:

$$\text{Exp Res}\ \dfrac{\text{Exp Let}\ \dfrac{D \rightsquigarrow^\emptyset [\Delta'' \mid D'] \qquad \dfrac{(1)}{\Gamma, a \updownarrow T'; \Delta_1, \Delta'' \vdash D' : U} \qquad \dfrac{(\Gamma; \Delta_2) \bullet x : U \vdash E' : V}{(\Gamma, a \updownarrow T'; \Delta_2) \bullet x : U \vdash E' : V}}{\Gamma, a \updownarrow T'; \Delta_A, \Delta' \vdash \mathsf{let}\ x = D\ \mathsf{in}\ E' : V}}{\Gamma; \Delta_A \vdash (\nu a)(\mathsf{let}\ x = E\ \mathsf{in}\ E') : V}$$

where (1) is constructed as follows:

$$\textsc{Exp Subsum}\ \dfrac{\dfrac{(2)}{\Gamma, a \updownarrow T'; \Delta_{11}, \Delta'' \vdash D' : U'} \qquad \dfrac{\Gamma; \Delta_{12} \vdash U' <: U}{\Gamma, a \updownarrow T'; \Delta_{12} \vdash U' <: U}}{\Gamma, a \updownarrow T'; \Delta_1, \Delta'' \vdash D' : U}$$

and (2) is derived from $\Gamma, a \updownarrow T'; \Delta_{11} \vdash D : U'$ and $D \leadsto^{\emptyset} [\Delta'' \mid D']$ using Lemma 3.2.42. We conclude $\Gamma; \Delta \vdash (\nu a)(\mathsf{let}\ x = E\ \mathsf{in}\ E') : T$ by (Exp Subsum).

Case (Heat Fork Comm): assume $(E \upharpoonright E') \upharpoonright E'' \Rrightarrow (E' \upharpoonright E) \upharpoonright E''$ with $\Gamma; \Delta \vdash (E \upharpoonright E') \upharpoonright E'' : T$. The judgement must follow by an instance of (Exp Fork) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash (E \upharpoonright E') \upharpoonright E'' : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $E \upharpoonright E' \leadsto [\Delta_1, \Delta_2 \mid D_1 \upharpoonright D_2]$ with $E \leadsto [\Delta_1 \mid D_1]$ and $E' \leadsto [\Delta_2 \mid D_2]$
- $E'' \leadsto [\Delta_3 \mid D_3]$
- $\Gamma; \Delta_A, \Delta_1, \Delta_2, \Delta_3 \hookrightarrow \Gamma; \Delta'_1, \Delta'_2$
- $\Gamma; \Delta'_1 \vdash D_1 \upharpoonright D_2 : U$
- $\Gamma; \Delta'_2 \vdash D_3 : V$

Now we notice that $\Gamma; \Delta'_1 \vdash D_1 \upharpoonright D_2 : U$ must have been derived by an instance of (Exp Fork) after an instance of (Exp Subsum). Since $D_1 \leadsto [\emptyset \mid D_1]$ and $D_2 \leadsto [\emptyset \mid D_2]$ by Lemma 3.2.38, it must be the case that $\Gamma; \Delta'_{11} \vdash D_1 \upharpoonright D_2 : U_2$ and $\Gamma; \Delta'_{12} \vdash U_2 <: U$ with:

- $\Gamma; \Delta'_1 \hookrightarrow \Gamma; \Delta'_{11}, \Delta'_{12}$
- $\Gamma; \Delta'_{11} \hookrightarrow \Gamma; \Delta'_A, \Delta'_B$
- $\Gamma; \Delta'_A \vdash D_1 : U_1$
- $\Gamma; \Delta'_B \vdash D_2 : U_2$

We have $E' \upharpoonright E \leadsto [\Delta_1, \Delta_2 \mid D_2 \upharpoonright D_1]$ by applying (Extr Fork) to $E \leadsto [\Delta_1 \mid D_1]$ and $E' \leadsto [\Delta_2 \mid D_2]$, hence we can construct the following type derivation:

$$\textsc{Exp Fork}\ \dfrac{\textsc{Exp Fork}\ \dfrac{\Gamma; \Delta'_B \vdash D_2 : U_2 \qquad \Gamma; \Delta'_A \vdash D_1 : U_1}{\Gamma; \Delta'_B, \Delta'_A \vdash D_2 \upharpoonright D_1 : U_1} \qquad \Gamma; \Delta'_2 \vdash D_3 : V}{\Gamma; \Delta_A \vdash (E' \upharpoonright E) \upharpoonright E'' : V}$$

since $\Gamma; \Delta_A, \Delta_1, \Delta_2, \Delta_3 \hookrightarrow \Gamma; (\Delta'_B, \Delta'_A), \Delta'_2$ can be derived by Lemma 3.2.8. Finally, we conclude $\Gamma; \Delta \vdash (E' \upharpoonright E) \upharpoonright E'' : T$ by (Exp Subsum).

Case (Heat Fork Assoc): assume $(E \Rsh E') \Rsh E'' \Rrightarrow E \Rsh (E' \Rsh E'')$ with $\Gamma;\Delta \vdash (E \Rsh E') \Rsh E'' : T$. The judgement must follow by an instance of (Exp Fork) after an instance of (Exp Subsum), hence it must be the case that $\Gamma;\Delta_A \vdash (E \Rsh E') \Rsh E'' : V$ and $\Gamma;\Delta_B \vdash V <: T$ with:

- $\Gamma;\Delta \hookrightarrow \Gamma;\Delta_A,\Delta_B$
- $E \Rsh E' \rightsquigarrow [\Delta_1,\Delta_2 \mid D_1 \Rsh D_2]$ with $E \rightsquigarrow [\Delta_1 \mid D_1]$ and $E' \rightsquigarrow [\Delta_2 \mid D_2]$
- $E'' \rightsquigarrow [\Delta_3 \mid D_3]$
- $\Gamma;\Delta_A,\Delta_1,\Delta_2,\Delta_3 \hookrightarrow \Gamma;\Delta'_1,\Delta'_2$
- $\Gamma;\Delta'_1 \vdash D_1 \Rsh D_2 : U$
- $\Gamma;\Delta'_2 \vdash D_3 : V$

Now we notice that $\Gamma;\Delta'_1 \vdash D_1 \Rsh D_2 : U$ must have been derived by an instance of (Exp Fork) after an instance of (Exp Subsum). Since $D_1 \rightsquigarrow [\emptyset \mid D_1]$ and $D_2 \rightsquigarrow [\emptyset \mid D_2]$ by Lemma 3.2.38, it must be the case that $\Gamma;\Delta'_{11} \vdash D_1 \Rsh D_2 : U_2$ and $\Gamma;\Delta'_{12} \vdash U_2 <: U$ with:

- $\Gamma;\Delta'_1 \hookrightarrow \Gamma;\Delta'_{11},\Delta'_{12}$
- $\Gamma;\Delta'_{11} \hookrightarrow \Gamma;\Delta'_A,\Delta'_B$
- $\Gamma;\Delta'_A \vdash D_1 : U_1$
- $\Gamma;\Delta'_B \vdash D_2 : U_2$

We have $E' \Rsh E'' \rightsquigarrow [\Delta_2,\Delta_3 \mid D_2 \Rsh D_3]$ by applying (Extr Fork) to $E' \rightsquigarrow [\Delta_2 \mid D_2]$ and $E'' \rightsquigarrow [\Delta_3 \mid D_3]$. Moreover, we know that $E'' \rightsquigarrow [\Delta_3 \mid D_3]$ implies $D_3 \rightsquigarrow [\emptyset \mid D_3]$ by Lemma 3.2.38, hence we can construct the following type derivation:

$$\textsc{Exp Fork}\ \dfrac{\Gamma;\Delta'_A \vdash D_1 : U_1 \qquad \dfrac{\Gamma;\Delta'_B \vdash D_2 : U_2 \qquad \Gamma;\Delta'_2 \vdash D_3 : V}{\Gamma;\Delta'_B,\Delta'_2 \vdash D_2 \Rsh D_3 : V}\ \textsc{Exp Fork}}{\Gamma;\Delta_A \vdash E \Rsh (E' \Rsh E'') : V}$$

since $\Gamma;\Delta_A,\Delta_1,\Delta_2,\Delta_3 \hookrightarrow \Gamma;\Delta'_A,(\Delta'_B,\Delta'_2)$ can be derived by Lemma 3.2.8. Finally, we conclude $\Gamma;\Delta \vdash E \Rsh (E' \Rsh E'') : T$ by (Exp Subsum).

Case (Heat Fork Let): assume $\mathsf{let}\ x = (E \Rsh E')\ \mathsf{in}\ E'' \Rrightarrow E \Rsh (\mathsf{let}\ x = E'\ \mathsf{in}\ E'')$ with $\Gamma;\Delta \vdash \mathsf{let}\ x = (E \Rsh E')\ \mathsf{in}\ E'' : T$. The judgement must follow by an instance of (Exp Let) after an instance of (Exp Subsum), hence it must be the case that $\Gamma;\Delta_A \vdash \mathsf{let}\ x = (E \Rsh E')\ \mathsf{in}\ E'' : V$ and $\Gamma;\Delta_B \vdash V <: T$ with:

- $\Gamma;\Delta \hookrightarrow \Gamma;\Delta_A,\Delta_B$
- $E \Rsh E' \rightsquigarrow [\Delta_1,\Delta_2 \mid D_1 \Rsh D_2]$ with $E \rightsquigarrow [\Delta_1 \mid D_1]$ and $E' \rightsquigarrow [\Delta_2 \mid D_2]$
- $\Gamma;\Delta_A,\Delta_1,\Delta_2 \hookrightarrow \Gamma;\Delta'_1,\Delta'_2$

- $\Gamma; \Delta'_1 \vdash D_1 \Rsh D_2 : U$
- $(\Gamma; \Delta'_2) \bullet x : U \vdash E'' : V$

Now we notice that $\Gamma; \Delta'_1 \vdash D_1 \Rsh D_2 : U$ must have been derived by an instance of (EXP FORK) after an instance of (EXP SUBSUM). Since $D_1 \leadsto [\emptyset \mid D_1]$ and $D_2 \leadsto [\emptyset \mid D_2]$ by Lemma 3.2.38, it must be the case that $\Gamma; \Delta'_{11} \vdash D_1 \Rsh D_2 : U_2$ and $\Gamma; \Delta'_{12} \vdash U_2 <: U$ with:

- $\Gamma; \Delta'_1 \hookrightarrow \Gamma; \Delta'_{11}, \Delta'_{12}$
- $\Gamma; \Delta'_{11} \hookrightarrow \Gamma; \Delta'_A, \Delta'_B$
- $\Gamma; \Delta'_A \vdash D_1 : U_1$
- $\Gamma; \Delta'_B \vdash D_2 : U_2$

We have $\mathsf{let}\ x = E'\ \mathsf{in}\ E'' \leadsto [\Delta_2 \mid \mathsf{let}\ x = D_2\ \mathsf{in}\ E'']$ by (EXTR LET), hence we can construct the following type derivation:

$$\text{EXP FORK}\ \dfrac{\Gamma; \Delta'_A \vdash D_1 : U_1 \qquad \dfrac{\text{EXP SUBSUM}\ \dfrac{\Gamma; \Delta'_B \vdash D_2 : U_2 \qquad \Gamma; \Delta'_{12} \vdash U_2 <: U}{\Gamma; \Delta'_B, \Delta'_{12} \vdash D_2 : U} \qquad (\Gamma; \Delta'_2) \bullet x : U \vdash E'' : V}{\Gamma; \Delta'_B, \Delta'_{12}, \Delta'_2 \vdash \mathsf{let}\ x = D_2\ \mathsf{in}\ E'' : V}\ \text{EXP LET}}{\Gamma; \Delta_A \vdash E \Rsh (\mathsf{let}\ x = E'\ \mathsf{in}\ E'') : V}$$

since $\Gamma; \Delta_A, \Delta_1, \Delta_2 \hookrightarrow \Gamma; \Delta'_A, (\Delta'_B, \Delta'_{12}, \Delta'_2)$ can be derived by Lemma 3.2.8. We conclude $\Gamma; \Delta \vdash E \Rsh (\mathsf{let}\ x = E'\ \mathsf{in}\ E'') : T$ by (EXP SUBSUM).

Assume now $E \Rsh (\mathsf{let}\ x = E'\ \mathsf{in}\ E'') \Rrightarrow \mathsf{let}\ x = (E \Rsh E')\ \mathsf{in}\ E''$ with $\Gamma; \Delta \vdash E \Rsh (\mathsf{let}\ x = E'\ \mathsf{in}\ E'') : T$. The judgement must follow by an instance of (EXP FORK) after an instance of (EXP SUBSUM), hence it must be the case that $\Gamma; \Delta_A \vdash E \Rsh (\mathsf{let}\ x = E'\ \mathsf{in}\ E'') : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $E \leadsto [\Delta_1 \mid D_1]$
- $\mathsf{let}\ x = E'\ \mathsf{in}\ E'' \leadsto [\Delta_2 \mid \mathsf{let}\ x = D_2\ \mathsf{in}\ E'']$ with $E' \leadsto [\Delta_2 \mid D_2]$
- $\Gamma; \Delta_A, \Delta_1, \Delta_2 \hookrightarrow \Gamma; \Delta'_1, \Delta'_2$
- $\Gamma; \Delta'_1 \vdash D_1 : U_1$
- $\Gamma; \Delta'_2 \vdash \mathsf{let}\ x = D_2\ \mathsf{in}\ E'' : V$

Now we notice that $\Gamma; \Delta'_2 \vdash \mathsf{let}\ x = D_2\ \mathsf{in}\ E'' : V$ must have been derived by an instance of (EXP LET) after an instance of (EXP SUBSUM). Since $E' \leadsto [\Delta_2 \mid D_2]$ implies $D_2 \leadsto [\emptyset \mid D_2]$ by Lemma 3.2.38, it must be the case that $\Gamma; \Delta'_{21} \vdash \mathsf{let}\ x = D_2\ \mathsf{in}\ E'' : U_3$ and $\Gamma; \Delta'_{22} \vdash U_3 <: V$ with:

- $\Gamma; \Delta'_2 \hookrightarrow \Gamma; \Delta'_{21}, \Delta'_{22}$

- $\Gamma; \Delta'_{21} \hookrightarrow \Gamma; \Delta'_A, \Delta'_B$
- $\Gamma; \Delta'_A \vdash D_2 : U_2$
- $(\Gamma; \Delta'_B) \bullet x : U_2 \vdash E'' : U_3$

We have $E \upharpoonleft E' \rightsquigarrow [\Delta_1, \Delta_2 \mid D_1 \upharpoonleft D_2]$ by (Extr Fork). Moreover, we know that $E \rightsquigarrow [\Delta_1 \mid D_1]$ implies $D_1 \rightsquigarrow [\emptyset \mid D_1]$ by Lemma 3.2.38, hence we can construct the following type derivation:

$$\text{Exp Let}\ \dfrac{\text{Exp Fork}\ \dfrac{\Gamma; \Delta'_1 \vdash D_1 : U_1 \qquad \Gamma; \Delta'_A \vdash D_2 : U_2}{\Gamma; \Delta'_1, \Delta'_A \vdash D_1 \upharpoonleft D_2 : U_2} \qquad \dfrac{(1)}{(\Gamma; \Delta'_B, \Delta'_{22}) \bullet x : U_2 \vdash E'' : V}}{\Gamma; \Delta_A \vdash \mathsf{let}\ x = (E \upharpoonleft E')\ \mathsf{in}\ E'' : V}$$

where $\Gamma; \Delta_A, \Delta_1, \Delta_2 \hookrightarrow \Gamma; (\Delta'_1, \Delta'_A), (\Delta'_B, \Delta'_{22})$ can be derived by Lemma 3.2.8, and the derivation (1) is constructed as follows:

$$\text{Exp Subsum}\ \dfrac{(\Gamma; \Delta'_B) \bullet x : U_2 \vdash E'' : U_3 \qquad \dfrac{\Gamma; \Delta'_{22} \vdash U_3 <: V}{\Gamma, x : \psi(U_2); \Delta'_{22} \vdash U_3 <: V}}{(\Gamma; \Delta'_B, \Delta'_{22}) \bullet x : U_2 \vdash E'' : V}$$

We conclude $\Gamma; \Delta \vdash \mathsf{let}\ x = (E \upharpoonleft E')\ \mathsf{in}\ E'' : T$ by (Exp Subsum).

□

Lemma 3.2.46 (Removing Tautologies). *If* $\Gamma; \Delta, F \vdash E : T$ *and* $\emptyset \vdash F$, *then* $\Gamma; \Delta \vdash E : T$.

Proof. We know that $\Gamma; \Delta, F \vdash E : T$ implies $\Gamma; \Delta, F \vdash \diamond$ by Lemma 3.2.5. Moreover, the latter implies $\Gamma; \Delta \vdash \diamond$ again by Lemma 3.2.5. Since $\Delta, F \vdash \Delta, F$ by Lemma 3.2.2, we can derive $\Gamma; \Delta \hookrightarrow \Gamma; \Delta, F$ as follows:

$$\text{Rewrite}\ \dfrac{\Gamma; \Delta \vdash \diamond \qquad \text{Cut}\ \dfrac{\emptyset \vdash F \qquad \Delta, F \vdash \Delta, F}{\Delta \vdash \Delta, F} \qquad \Gamma; \Delta, F \vdash \diamond}{\Gamma; \Delta \hookrightarrow \Gamma; \Delta, F}$$

Since $\Gamma; \Delta, F \vdash E : T$, the conclusion $\Gamma; \Delta \vdash E : T$ follows by Lemma 3.2.9. □

Theorem 3.2.47 (Subject Reduction). *Let* $fv(E) = \emptyset$. *If* $\Gamma; \Delta \vdash E : T$ *and* $E \to E'$, *then* $\Gamma; \Delta \vdash E' : T$.

Proof. By induction on the derivation of $E \to E'$. In the proof we implicitly appeal to Lemma 3.2.5 and Lemma 3.2.8 several times:

Case (Red Fun): assume $(\lambda x. E)\ N \to E\{N/x\}$ and $\Gamma; \Delta \vdash (\lambda x. E)\ N : T$. The typing judgement must follow by an instance of (Exp Appl) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash (\lambda x. E)\ N : U'\{N/x\}$ and $\Gamma; \Delta_B \vdash U'\{N/x\} <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $\Gamma; \Delta_A \hookrightarrow \Gamma; \Delta_1, \Delta_2$
- $\Gamma; \Delta_1 \vdash \lambda x. E : x : T' \to U'$
- $\Gamma; \Delta_2 \vdash N : T'$

By Lemma 3.2.31 we know that $\Gamma; \Delta_1 \vdash \lambda x. E : x : T' \to U'$ implies $(\Gamma; \Delta_1) \bullet x : T' \vdash E : U'$. Now notice that $x \notin \mathit{dom}(\Gamma)$ by Lemma 3.2.5, hence $x \notin \mathit{fv}(\Delta_1)$ by Lemma 3.2.10. By applying Lemma 3.2.27, we then get $\Gamma; \Delta_1, \Delta_2 \vdash E\{N/x\} : U'\{N/x\}$. Since $\Gamma; \Delta \hookrightarrow \Gamma; (\Delta_1, \Delta_2), \Delta_B$, the conclusion $\Gamma; \Delta \vdash E\{N/x\} : T$ follows by an application of (Exp Subsum).

Case (Red Split): assume $\mathsf{let}\ (x, y) = (M, N)\ \mathsf{in}\ E \to E\{M/x\}\{N/y\}$ and $\Gamma; \Delta \vdash \mathsf{let}\ (x, y) = (M, N)\ \mathsf{in}\ E : T$. The typing judgement must follow by an instance of (Exp Split) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash \mathsf{let}\ (x, y) = (M, N)\ \mathsf{in}\ E : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $\Gamma; \Delta_A \hookrightarrow \Gamma; \Delta_1, \Delta_2$
- $\Gamma; \Delta_1 \vdash (M, N) : x : T' * U'$
- $(\Gamma; \Delta_2) \bullet x : T' \bullet y : U' \bullet !((x, y) = (M, N)) \vdash E : V$
- $\{x, y\} \cap \mathit{fv}(V) = \emptyset$

By Lemma 3.2.32 we know that $\Gamma; \Delta_1 \vdash (M, N) : x : T' * U'$ implies:

- $\Gamma; \Delta_1 \hookrightarrow \Gamma; \Delta_{11}, \Delta_{12}$
- $\Gamma; \Delta_{11} \vdash M : T'$
- $\Gamma; \Delta_{12} \vdash N : U'\{M/x\}$

Now notice that $x \notin \mathit{dom}(\Gamma)$ by Lemma 3.2.5, hence $x \notin \mathit{fv}(\Delta_2)$ by Lemma 3.2.10. By applying Lemma 3.2.27 twice and noting that $\{x, y\} \cap \mathit{fv}(V) = \emptyset$, we then get $\Gamma; \Delta_{11}, \Delta_{12}, \Delta_2, !((M, N) = (M, N)) \vdash E : V$. Since $\emptyset \vdash !((M, N) = (M, N))$, the latter judgement implies $\Gamma; \Delta_{11}, \Delta_{12}, \Delta_2 \vdash E : V$ by Lemma 3.2.46. Since $\Gamma; \Delta \hookrightarrow \Gamma; (\Delta_{11}, \Delta_{12}, \Delta_2), \Delta_B$, the conclusion $\Gamma; \Delta \vdash E : T$ follows by (Exp Subsum).

Case (Red Match): assume $\mathsf{match}\ h\ N\ \mathsf{with}\ h\ x\ \mathsf{then}\ E\ \mathsf{else}\ E' \to E\{N/x\}$ and $\Gamma; \Delta \vdash \mathsf{match}\ h\ N\ \mathsf{with}\ h\ x\ \mathsf{then}\ E\ \mathsf{else}\ E' : T$. The typing judgement must follow by an instance of (Exp Match) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash \mathsf{match}\ h\ N\ \mathsf{with}\ h\ x\ \mathsf{then}\ E\ \mathsf{else}\ E' : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $\Gamma; \Delta_A \hookrightarrow \Gamma; \Delta_1, \Delta_2$

- $\Gamma; \Delta_1 \vdash h\ N : T'$
- $(\Gamma; \Delta_2) \bullet x : U' \bullet !(h\ x = h\ N) \vdash E : V$
- $\Gamma; \Delta_2 \vdash E' : V$
- $(h, T', U') \in \{(\mathsf{inl}, T_1 + T_2, T_1), (\mathsf{inr}, T_1 + T_2, T_2), (\mathsf{fold}, \mu\alpha.\, T_1, T_1\{\mu\alpha.\, T_1/\alpha\})\}$

According to the form of h, we invoke either Lemma 3.2.33 or Lemma 3.2.34. and we get $\Gamma; \Delta_1 \vdash N : U'$. Now we notice that $\Gamma; \Delta_2 \vdash E' : V$ implies $\mathit{fnfv}(V) \subseteq \mathit{dom}(\Gamma)$ by Lemma 3.2.5, hence the fact that $x \notin \mathit{dom}(\Gamma)$ implies $x \notin \mathit{fv}(V)$. Moreover, $x \notin \mathit{dom}(\Gamma)$ implies $x \notin \mathit{fv}(\Delta_2)$ by Lemma 3.2.10. By applying Lemma 3.2.27 we then get $\Gamma; \Delta_1, \Delta_2, !(h\ N = h\ N) \vdash E\{N/x\} : V$. Since $\emptyset \vdash !(h\ N = h\ N)$, the latter judgement implies $\Gamma; \Delta_1, \Delta_2 \vdash E\{N/x\} : V$ by Lemma 3.2.46. Since $\Gamma; \Delta \hookrightarrow \Gamma; (\Delta_1, \Delta_2), \Delta_B$, the conclusion $\Gamma; \Delta \vdash E\{N/x\} : T$ follows by (Exp Subsum).

Assume now $\mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ E\ \mathsf{else}\ E' \to E'$ with $M \neq h\ N$ for all N. The type derivation has the same structure as before, but for the obvious changes. Since $\Gamma; \Delta_2 \vdash E' : V$ and $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_2, \Delta_B$, the conclusion $\Gamma; \Delta \vdash E' : T$ follows by (Exp Subsum).

Case (Red Eq): assume we have $M = M \to \mathsf{true}$ and $\Gamma; \Delta \vdash M = M : T$. The typing judgement must follow by an instance of (Exp Eq) after an instance of (Exp Subsum), hence it must be the case that:

$$\Gamma; \Delta_A \vdash M = M : \{x : \mathsf{bool} \mid !(x = \mathsf{true} \multimap M = M)\}$$

and:

$$\Gamma; \Delta_B \vdash \{x : \mathsf{bool} \mid !(x = \mathsf{true} \multimap M = M)\} <: T.$$

with $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$. Recall now that $\mathsf{true} \triangleq \mathsf{inl}(\,)$ and $\mathsf{bool} \triangleq \mathsf{unit} + \mathsf{unit}$, so it is easy to show that we have $\Gamma; \Delta_A \vdash \mathsf{true} : \mathsf{bool}$. Now we note that:

$$\Gamma; \emptyset \vdash !(\mathsf{true} = \mathsf{true} \multimap M = M),$$

thus we get $\Gamma; \Delta_A \vdash \mathsf{true} : \{x : \mathsf{bool} \mid !(x = \mathsf{true} \multimap M = M)\}$ by (Val Refine) and the conclusion $\Gamma; \Delta \vdash \mathsf{true} : T$ follows by an application of (Exp Subsum).

Assume, instead, that $M = N \to \mathsf{false}$ with $M \neq N$ and $\Gamma; \Delta \vdash M = N : T$. The typing judgement must follow by an instance of (Exp Eq) after an instance of (Exp Subsum), hence it must be the case that:

$$\Gamma; \Delta_A \vdash M = N : \{x : \mathsf{bool} \mid !(x = \mathsf{true} \multimap M = N)\}$$

and:

$$\Gamma; \Delta_B \vdash \{x : \mathsf{bool} \mid !(x = \mathsf{true} \multimap M = N)\} <: T.$$

with $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$. Now we note that:

$$\Gamma; \emptyset \vdash !(\mathsf{false} = \mathsf{true} \multimap M = N),$$

thus we get $\Gamma; \Delta_A \vdash \mathsf{false} : \{x : \mathsf{bool} \mid !(x = \mathsf{true} \multimap M = N)\}$ by (Val Refine) and the conclusion $\Gamma; \Delta \vdash \mathsf{false} : T$ follows by an application of (Exp Subsum).

Case (Red Comm): assume $a!M \Rsh a? \to M$ and $\Gamma; \Delta \vdash a!M \Rsh a? : T$. The typing judgement must follow by an instance of (Exp Fork) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash a!M \Rsh a? : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $a!M \rightsquigarrow [\emptyset \mid a!M]$
- $a? \rightsquigarrow [\emptyset \mid a?]$
- $\Gamma; \Delta_A \hookrightarrow \Gamma; \Delta_1, \Delta_2$
- $\Gamma; \Delta_1 \vdash a!M : U$
- $\Gamma; \Delta_2 \vdash a? : V$

We notice that $\Gamma; \Delta_1 \vdash a!M : U$ must follow by an instance of (Exp Send) after an instance of (Exp Subsum), hence:

- $\Gamma; \Delta_1 \hookrightarrow \Gamma; \Delta_{11}, \Delta_{12}$
- $\Gamma; \Delta_{11} \vdash a!M : \mathsf{unit}$
- $\Gamma; \Delta_{12} \vdash \mathsf{unit} <: U$
- $(a \updownarrow T') \in \Gamma$
- $\Gamma; \Delta_{11} \vdash M : T'$

We also notice that $\Gamma; \Delta_2 \vdash a? : V$ must follow by an instance of (Exp Recv) after an instance of (Exp Subsum), hence:

- $\Gamma; \Delta_2 \hookrightarrow \Gamma; \Delta_{21}, \Delta_{22}$
- $\Gamma; \Delta_{21} \vdash a? : T'$, since $(a \updownarrow T') \in \Gamma$
- $\Gamma; \Delta_{22} \vdash T' <: V$

Thus we get $\Gamma; \Delta_{11}, \Delta_{22} \vdash M : V$ by (Exp Subsum). Since $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_{11}, \Delta_{22}, \Delta_B$, the conclusion $\Gamma; \Delta \vdash M : T$ follows by an application of (Exp Subsum).

Case (Red Let Val): assume $\mathsf{let}\ x = M\ \mathsf{in}\ E \to E\{M/x\}$ and $\Gamma; \Delta \vdash \mathsf{let}\ x = M\ \mathsf{in}\ E : T$. The typing judgement must follow by an instance of (Exp Let) after an instance of (Exp Subsum). Notice that $M \rightsquigarrow [\emptyset \mid M]$, hence it must be the case that $\Gamma; \Delta_A \vdash \mathsf{let}\ x = M\ \mathsf{in}\ E : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $\Gamma; \Delta_A \hookrightarrow \Gamma; \Delta_1, \Delta_2$

- $\Gamma; \Delta_1 \vdash M : U$
- $(\Gamma; \Delta_2) \bullet x : U \vdash E : V$
- $x \notin \mathit{fv}(V)$

Now notice that $x \notin \mathit{dom}(\Gamma)$ by Lemma 3.2.5, hence $x \notin \mathit{fv}(\Delta_2)$ by Lemma 3.2.10. By applying Lemma 3.2.27 and noting that $x \notin \mathit{fv}(V)$, we then get $\Gamma; \Delta_1, \Delta_2 \vdash E\{M/x\} : V$. Since $\Gamma; \Delta \hookrightarrow \Gamma; (\Delta_1, \Delta_2), \Delta_B$, the conclusion $\Gamma; \Delta \vdash E\{M/x\} : T$ follows by an application of (Exp Subsum).

Case (Red Let): assume $\mathsf{let}\ x = E\ \mathsf{in}\ E'' \to \mathsf{let}\ x = E'\ \mathsf{in}\ E''$ with $E \to E'$ and $\Gamma; \Delta \vdash \mathsf{let}\ x = E\ \mathsf{in}\ E'' : T$. The typing judgement must follow by an instance of (Exp Let) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash \mathsf{let}\ x = E\ \mathsf{in}\ E'' : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $E \rightsquigarrow [\Delta' \mid D]$
- $\Gamma; \Delta_A, \Delta' \hookrightarrow \Gamma; \Delta_1, \Delta_2$
- $\Gamma; \Delta_1 \vdash D : U$
- $(\Gamma; \Delta_2) \bullet x : U \vdash E'' : V$

By Lemma 3.2.39 we know that $E \to E'$ and $E \rightsquigarrow [\Delta' \mid D]$ imply that there exist D', Δ'', D'', D^* such that $D \to D'$ and $E' \rightsquigarrow [\Delta', \Delta'' \mid D'']$ with $D' \rightsquigarrow [\Delta'' \mid D^*]$ and $D^* \Rrightarrow D''$. Since Lemma 3.2.39 is depth-preserving, we can apply the inductive hypothesis and get $\Gamma; \Delta_1 \vdash D' : U$. Given that $D' \rightsquigarrow [\Delta'' \mid D^*]$ and $\Gamma; \Delta_1 \vdash D' : U$, we get $\Gamma; \Delta_1, \Delta'' \vdash D^* : U$ by Lemma 3.2.42. Since $D^* \Rrightarrow D''$ and $\Gamma; \Delta_1, \Delta'' \vdash D^* : U$, we get $\Gamma; \Delta_1, \Delta'' \vdash D'' : U$ by Lemma 3.2.45. Hence, we have:

- $E' \rightsquigarrow [\Delta', \Delta'' \mid D'']$
- $\Gamma; \Delta_A, \Delta', \Delta'' \hookrightarrow \Gamma; (\Delta_1, \Delta''), \Delta_2$
- $\Gamma; \Delta_1, \Delta'' \vdash D'' : U$
- $(\Gamma; \Delta_2) \bullet x : U \vdash E'' : V$

We can then apply rule (Exp Let) to get $\Gamma; \Delta_A \vdash \mathsf{let}\ x = E'\ \mathsf{in}\ E'' : V$. The conclusion $\Gamma; \Delta \vdash \mathsf{let}\ x = E'\ \mathsf{in}\ E'' : T$ follows by (Exp Subsum).

Case (Red Res): assume $(\nu a)E \to (\nu a)E'$ with $E \to E'$ and $\Gamma; \Delta \vdash (\nu a)E : T$. The typing judgement must follow by an instance of (Exp Res) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash (\nu a)E : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $E \rightsquigarrow^a [\Delta' \mid D]$

- $\Gamma, a \updownarrow U; \Delta_A, \Delta' \vdash D : V$

By Lemma 3.2.39 we know that $E \to E'$ and $E \leadsto^a [\Delta' \mid D]$ imply that there exist D', Δ'', D'', D^* such that $D \to D'$ and $E' \leadsto^a [\Delta', \Delta'' \mid D'']$ with $D' \leadsto^a [\Delta'' \mid D^*]$ and $D^* \Rrightarrow D''$. Since Lemma 3.2.39 is depth-preserving, we can apply the inductive hypothesis and get $\Gamma, a \updownarrow U; \Delta_A, \Delta' \vdash D' : V$. Given that $D' \leadsto^a [\Delta'' \mid D^*]$ and $\Gamma, a \updownarrow U; \Delta_A, \Delta' \vdash D' : V$, we get $\Gamma, a \updownarrow U; \Delta_A, \Delta', \Delta'' \vdash D^* : V$ by Lemma 3.2.42. By Lemma 3.2.45 we get $\Gamma, a \updownarrow U; \Delta_A, \Delta', \Delta'' \vdash D'' : V$. Hence, we have:

- $E' \leadsto^a [\Delta', \Delta'' \mid D'']$
- $\Gamma, a \updownarrow U; \Delta_A, \Delta', \Delta'' \vdash D'' : V$

We can then apply rule (Exp Res) to get $\Gamma; \Delta_A \vdash (\nu a)E' : V$. The conclusion $\Gamma; \Delta \vdash (\nu a)E' : T$ follows by (Exp Subsum).

Case (Red Fork 1): assume $E \upharpoonright E'' \to E' \upharpoonright E''$ with $E \to E'$ and $\Gamma; \Delta \vdash E \upharpoonright E'' : T$. The typing judgement must follow by an instance of (Exp Fork) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash E \upharpoonright E'' : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $E \leadsto [\Delta' \mid D_1]$
- $E'' \leadsto [\Delta'' \mid D_2]$
- $\Gamma; \Delta_A, \Delta', \Delta'' \hookrightarrow \Gamma; \Delta_1, \Delta_2$
- $\Gamma; \Delta_1 \vdash D_1 : U$
- $\Gamma; \Delta_2 \vdash D_2 : V$

By Lemma 3.2.39 we know that $E \to E'$ and $E \leadsto [\Delta' \mid D_1]$ imply that there exist D_1', Δ^*, D'', D^* such that $D_1 \to D_1'$ and $E' \leadsto [\Delta', \Delta^* \mid D'']$ with $D_1' \leadsto [\Delta^* \mid D^*]$ and $D^* \Rrightarrow D''$. Since Lemma 3.2.39 is depth-preserving, we can apply the inductive hypothesis and get $\Gamma; \Delta_1 \vdash D_1' : U$. Given that $D_1' \leadsto [\Delta^* \mid D^*]$ and $\Gamma; \Delta_1 \vdash D_1' : U$, we get $\Gamma; \Delta_1, \Delta^* \vdash D^* : U$ by Lemma 3.2.42. By Lemma 3.2.45 we get $\Gamma; \Delta_1, \Delta^* \vdash D'' : U$. Hence, we have:

- $E' \leadsto [\Delta', \Delta^* \mid D'']$
- $E'' \leadsto [\Delta'' \mid D_2]$
- $\Gamma; \Delta_A, \Delta', \Delta^*, \Delta'' \hookrightarrow \Gamma; (\Delta_1, \Delta^*), \Delta_2$
- $\Gamma; \Delta_1, \Delta^* \vdash D'' : U$
- $\Gamma; \Delta_2 \vdash D_2 : V$

We can then apply rule (Exp Fork) to get $\Gamma; \Delta_A \vdash E' \upharpoonright E'' : V$. The conclusion $\Gamma; \Delta \vdash E' \upharpoonright E'' : T$ follows by (Exp Subsum).

Case (Red Fork 2): analogous to the previous case.

Case (Red Heat): assume $E \to E'$ with $E \Rrightarrow D$, $D \to D'$ and $D' \to E'$. Assume further that $\Gamma; \Delta \vdash E : T$. By Lemma 3.2.45 we have $\Gamma; \Delta \vdash D : T$. By inductive hypothesis $\Gamma; \Delta \vdash D' : T$, hence $\Gamma; \Delta \vdash E' : T$ again by Lemma 3.2.45.

□

3.2.7 Proof of (robust) safety

Lemma 3.2.48 (Static Safety)**.** *If $\varepsilon; \emptyset \vdash \mathbf{S} : T$, then $\mathbf{S}$ is statically safe.*

Proof. Consider an arbitrary structure:

$$(\nu a_1) \dots (\nu a_r)((E_1 \upharpoonright E_2) \upharpoonright E_3) \upharpoonright E_4,$$

where:

- $E_1 = \Pi_{i \in [1,m]} \mathsf{assume}\ F_i$,
- $E_2 = \Pi_{j \in [1,n]} \mathsf{assert}\ F'_j$,
- $E_3 = \Pi_{k \in [1,o]} c_k!M_k$, and
- $E_4 = \Pi_{\ell \in [1,p]} \mathcal{L}_\ell[e_\ell]$.

We need to show that $F_1, \dots, F_m \vdash F'_1 \otimes \dots \otimes F'_n$.

We know that $\varepsilon; \emptyset \vdash \mathbf{S} : T$. This must have been derived by r applications of (Exp Res) followed by three applications of (Exp Fork), possibly interleaving with multiple applications of (Exp Subsum). Note that each application of (Exp Res) and (Exp Fork) will make use of extraction, but by Lemma 3.2.43 we can simplify an arbitrary chain of extraction steps with decreasing index sets $\{a_1, \dots, a_r\}, \dots, \{a_r\}, \emptyset$ into a single extraction step with index set $\emptyset$. Note that, by definition, extraction does not affect E_2, E_3, and E_4, since they do not contain assumptions, but extracts all the assumed formulas $F_i \neq \mathbf{1}$ from E_1. Also note that repeatedly extracting with the same index set $\emptyset$ does not yield any new result, as can be seen using Lemma 3.2.38. By transitivity of subtyping and rewriting, using the previous facts, without loss of generality we have:

- $((E_1 \upharpoonright E_2) \upharpoonright E_3) \upharpoonright E_4 \leadsto^{\emptyset} [\Delta_1 \mid ((D_1 \upharpoonright E_2) \upharpoonright E_3) \upharpoonright E_4]$,
- where $E_1 \leadsto^{\emptyset} [\Delta_1 \mid D_1]$ with $\Delta_1 = \{F_i \mid F_i \neq \mathbf{1}\}$ and $D_1 = \Pi_{i \in [1,m]} \mathsf{assume}\ \mathbf{1}$.
- $\Gamma; \Delta_1 \hookrightarrow \Gamma; (\Delta_{A_1}, \Delta_{A_2}, \Delta_{A_3}, \Delta_{A_4}), \Delta_B$ with $\Gamma = a_1 \updownarrow T_1, \dots, a_r \updownarrow T_r$
- $\Gamma; \Delta_{A_1} \vdash D_1 : U_1$ and $\Gamma; \Delta_{A_i} \vdash E_i : U_i$ for all $i \in \{2, 3, 4\}$
- $\Gamma; \Delta_B \vdash U_4 <: T$.

Hence, we know that $\Gamma; \Delta_{A_2} \vdash E_2 : U_2$, where E_2 is the parallel composition of the top-level assertions of **S**. Such a typing derivation must contain $n-1$ applications of (Exp-Fork) and n applications of (Exp Assert), possibly interleaved with multiple applications of (Exp Subsum). Again without loss of generality we have:

- $\Gamma; \Delta_{A_2} \hookrightarrow \Gamma; (\Delta_{C_1}, \ldots, \Delta_{C_n}), \Delta_D$
- for all $j \in \{1, \ldots, n\}$: $\Gamma; \Delta_{C_j} \vdash \mathsf{assert}\ F'_j : V_j$
- for all $j \in \{1, \ldots, n\}$: $\Gamma; \Delta_{C_j} \hookrightarrow \Gamma; \Delta'_{C_j}, \Delta''_{C_j}$ for some $\Delta'_{C_j}, \Delta''_{C_j}$ such that:
  - $\Gamma; \Delta'_{C_j} \vdash F'_j$, and
  - $\Gamma; \Delta''_{C_j} \vdash \mathsf{unit} <: V_j$
- $\Gamma; \Delta_D \vdash V_n <: U_2$.

By applying ($\otimes$-Right) and rule (Derive), it follows that:

$$\Gamma; \Delta'_{C_1}, \ldots, \Delta'_{C_n} \vdash F'_1 \otimes \ldots \otimes F'_n.$$

Using Lemma 3.2.8 we get $\Gamma; \Delta_1 \hookrightarrow \Gamma; \Delta'_{C_1}, \ldots, \Delta'_{C_n}$. By Lemma 3.2.9 it follows that $\Gamma; \Delta_1 \vdash F'_1 \otimes \ldots \otimes F'_n$. Since $\Delta_1 = \{F_i \mid F_i \neq \mathbf{1}\}$, we get $\Gamma; F_1, \ldots, F_m \vdash F'_1 \otimes \ldots \otimes F'_n$ by Lemma 3.2.7. By inverting rule (Derive) this implies:

$$F_1, \ldots, F_m \vdash F'_1 \otimes \ldots \otimes F'_n.$$

□

Restatement of Theorem 2.6.1. If $\varepsilon; \emptyset \vdash E : T$, then E is safe.

Proof. In order to prove that E is safe it suffices to show that, for all expressions E' and structures **S** such that $E \to^* E'$ and $E' \Rrightarrow \mathbf{S}$, it holds that **S** is statically safe.

By Theorem 3.2.47, $\varepsilon; \emptyset \vdash E : T$ implies $\varepsilon; \emptyset \vdash E' : T$. By Lemma 3.2.45, $E' \Rrightarrow \mathbf{S}$ implies $\varepsilon; \emptyset \vdash \mathbf{S} : T$. We can conclude that **S** is statically safe by Lemma 3.2.48. □

Lemma 3.2.49 (Universal Type). *If $\Gamma; \emptyset \vdash \diamond$, then $\Gamma; \emptyset \vdash T <:> \mathsf{Un}$ for all $T \in \{\mathsf{unit}, x : \mathsf{Un} \to \mathsf{Un}, x : \mathsf{Un} * \mathsf{Un}, \mathsf{Un} + \mathsf{Un}, \mu\alpha.\, \mathsf{Un}\}$.*

Proof. By inspection of the syntax-driven kinding rules it follows immediately that $\Gamma; \emptyset \vdash T :: k$ for all $T \in \{\mathsf{unit}, x : \mathsf{Un} \to \mathsf{Un}, x : \mathsf{Un} * \mathsf{Un}, \mathsf{Un} + \mathsf{Un}, \mu\alpha.\, \mathsf{Un}\}$ and $k \in \{\mathsf{pub}, \mathsf{tnt}\}$. We can then conclude by applying Lemma 3.2.19. □

Lemma 3.2.50 (Opponent Typability). *Let $\Gamma; \emptyset \vdash \diamond$. Let O be an expression that does not contain any assumption or assertion such that $(a \updownarrow \mathsf{Un}) \in \Gamma$ for each $a \in fn(O)$ and $(x : \mathsf{Un}) \in \Gamma$ for each $x \in fv(O)$, then $\Gamma; \emptyset \vdash O : \mathsf{Un}$.*

Proof. By induction on the structure of O. In each case we apply the value/expression typing rule corresponding to the structure of O (applying the induction hypothesis to the premises of the typing rule whenever needed). This allows us to derive that $\Gamma; \Delta \vdash O : T$ for some $T \in \{\mathsf{unit}, x : \mathsf{Un} \to \mathsf{Un}, x : \mathsf{Un} * \mathsf{Un}, \mathsf{Un} + \mathsf{Un}, \mu\alpha.\, \mathsf{Un}\}$ by using the following strategies:

- We first note that $O \leadsto^{\widetilde{a}} [\emptyset \mid O]$ for any $\widetilde{a}$, since by definition O does not contain any assumption.
- In the case of typing a constructor $h \in \{\mathsf{inl}, \mathsf{inr}\}$, we choose the "free" type to be Un.
- If O is of the form $M = N$, we additionally apply (Exp Subsum) with subtyping rule (Sub Refine).
- If O is a split or a match operation, we appeal to Lemma 3.2.7.
- We can easily switch between $T \in \{\mathsf{unit}, x : \mathsf{Un} \to \mathsf{Un}, x : \mathsf{Un} * \mathsf{Un}, \mathsf{Un} + \mathsf{Un}, \mu\alpha.\, \mathsf{Un}\}$ and Un by Lemma 3.2.49, using (Exp Subsum) whenever needed.

We conclude by an application of (Exp Subsum), using Lemma 3.2.49. □

Restatement of Theorem 2.6.2. If $\varepsilon; \emptyset \vdash E : \mathsf{Un}$, then E is robustly safe.

Proof. Consider an arbitrary opponent O, we need to show that the application $O\ E$ is safe. Recall that:

$$O\ E \triangleq \mathsf{let}\ f = O\ \mathsf{in}\ \mathsf{let}\ x = E\ \mathsf{in}\ f\ x.$$

Let $\Gamma = a_1 \updownarrow \mathsf{Un}, \ldots, a_n \updownarrow \mathsf{Un}$ with $fn(O) = \{a_1, \ldots, a_n\}$. Since the opponent O is closed by definition, by Lemma 3.2.50 we know that $\Gamma; \emptyset \vdash O : \mathsf{Un}$. We can apply (Exp Subsum) and Lemma 3.2.49 to derive:

$$\Gamma; \emptyset \vdash O : \mathsf{Un} \to \mathsf{Un}. \tag{3.1}$$

We can apply Lemma 3.2.7 to $\varepsilon; \emptyset \vdash E : \mathsf{Un}$ and get $\Gamma; \emptyset \vdash E : \mathsf{Un}$. Assume now $E \leadsto [\Delta \mid D]$, by Lemma 3.2.42 we have $\Gamma; \Delta \vdash D : \mathsf{Un}$. By Lemma 3.2.7 we then get:

$$\Gamma, f : \mathsf{Un} \to \mathsf{Un}; \Delta \vdash D : \mathsf{Un}. \tag{3.2}$$

Since $O \leadsto^{\emptyset} [\emptyset \mid O]$, we can construct the following type derivation:

$$\dfrac{(3.1)\ \dfrac{\cdots}{\Gamma;\emptyset\vdash O:\mathsf{Un}\to\mathsf{Un}} \qquad \dfrac{(3.2)\ \dfrac{\cdots}{\Gamma,f:\mathsf{Un}\to\mathsf{Un};\Delta\vdash D:\mathsf{Un}} \qquad \dfrac{\cdots}{\Gamma, f:\mathsf{Un}\to\mathsf{Un}, x:\mathsf{Un};\emptyset\vdash f\ x:\mathsf{Un}}\ \textsc{Exp Appl}}{\Gamma,f:\mathsf{Un}\to\mathsf{Un};\emptyset\vdash \mathsf{let}\ x=E\ \mathsf{in}\ f\ x:\mathsf{Un}}\ \textsc{Exp Let}}{\Gamma;\emptyset\vdash \mathsf{let}\ f=O\ \mathsf{in}\ \mathsf{let}\ x=E\ \mathsf{in}\ f\ x:\mathsf{Un}}\ \textsc{Exp Let}$$

Since $O\ E \leadsto^{\widetilde{b}} [\emptyset \mid O\ E]$ for all $\widetilde{b}$, we can get $\varepsilon; \emptyset \vdash (\nu a_1) \ldots (\nu a_n)(O\ E) : \mathsf{Un}$ by applying n times rule (Exp Res) to the conclusion of the derivation above. By Theorem 2.6.1, we then know that $(\nu a_1) \ldots (\nu a_n)(O\ E)$ is safe. Since restrictions do not affect safety, we can conclude. □

Chapter 4

Secure Access Control for Android Applications

4.1 Introduction

Mobile phones have quickly evolved, over the past few years, from simple devices intended for phone calls and text messaging, to powerful handheld PDAs, hosting sophisticated applications that manage personal data and interact on-line to share information and access (security-sensitive) services.

This evolution has attracted the interest of a growing community of researchers on mobile phone security, and on Android security in particular. Fundamental weaknesses and subtle design flaws of the Android architecture have been identified, studied and fixed. Originated with the seminal work in [34], a series of papers have developed techniques to ensure various system-level information-flow properties, by means of data-flow analysis [44], runtime detection mechanisms [32] and changes to the operating system [43]. Other papers have applied those same techniques in the study of application-level properties associated with Android's intent-based communication model and its interaction with the underlying permission system [29, 18]. Somewhat surprisingly, typing techniques have instead received very limited attention, with few notable exceptions to date ([25], and more recently [6]). Thus, the potential extent and scope of type-based analysis has been left largely unexplored.

In the present chapter, we make a step towards filling this gap, by developing a calculus to reason on the Android inter-component communication API, and a type system to statically analyze and control the interaction between intent-based communication and the underlying permission system. Our analysis of the Android platform is targeted at the static detection of *privilege escalation* attacks, a vulnerability which exposes the risk of unauthorized permission usage by malicious Android applications. Our goal is *enforcing* a more secure access control policy for well-typed Android applications, providing the expected security invariant that a component protected by a permission `P` can be influenced only by applications owning the permission `P`.

Contributions Though the problem of privilege escalation attacks on Android has been studied before [37, 18], we are the first to devise a static detection technique. To carry out our study, we introduce λ-`Perms`, a simple formal calculus for reasoning about the Android inter-component interaction. Albeit small and abstract, λ-`Perms` captures all the relevant aspects of the Android message passing architecture and its relationships with the underlying permission system. Interestingly, our approach pays off, as it allows us to unveil subtle attack surfaces to the current Android implementation that had not been observed by previous work.

We tackle the problem of programmatically preventing privilege escalation attacks inside λ-`Perms`, by spelling out a formal definition of safety and proposing a sound security type system which statically enforces such notion, despite the best efforts of an active opponent. Our safety definition is inspired by run-time mechanisms proposed in earlier work [37], but more compact and effective for formal reasoning. Enforcing the desired protection turns out to be challenging, because the inadvertent disclosure of sensitive data may enable some typically overlooked privilege escalation scenarios. Given that an opponent may actively try to fool well-typed components into revealing secret data, our type system must deal with both secrecy and authenticity issues to be proven sound. Our type discipline then provides formal assurance about some secure communication guidelines proposed in [26].

Based on our formal framework, we then develop a prototype implementation of `Lintent`, a type-based analyzer integrated with the Android Development Tools suite (ADT). `Lintent` integrates our typing technique for privilege escalation detection within a full-fledged static analysis framework that includes intent type reconstruction, manifest permission analysis, and a suite of other actions directed towards assisting the programmer in writing more robust and reliable applications. Enhancing the Android development process is increasingly being recognized as an urgent need [26, 36, 33, 58, 31]: `Lintent` represents a first step in that direction.

Structure of the chapter Section 4.2 reviews the basics of the Android architecture. Section 4.3 introduces λ-`Perms` and discusses its relationships with Android. Section 4.4 describes privilege escalation attacks. Section 4.5 presents a type-and-effect system to enforce protection against such attacks. Section 4.6 describes `Lintent` and details practical remarks. Section 4.7 discusses related work. All the proofs are presented in Chapter 5.

4.2 Android overview

We review the most important aspects of the Android architecture and its security model, thus providing the necessary ingredients to understand the technical contents of the chapter.

Intents Once installed on a device, Android applications run isolated from each other in their own security sandbox. Data and functionality sharing among different applications is implemented through a message-passing paradigm built on top of *intents*, i.e., passive data structures providing an abstract description of an operation to be performed and the associated parameters. For instance, an application can send an intent to an image viewer, requesting to display a given JPEG file, to avoid the need of internally reimplementing such functionality.

The most interesting aspect of intents is that they can be used for both *explicit* and *implicit* communication. Explicit intents specify their intended receiver by name and are always securely delivered to it; since the identity of the recipient is typically unknown to developers of third-party applications, explicit intents are particularly useful for intra-application communication. Implicit intents, instead, do not mention any specific receiver and just require delivery to any application that declares to support a given operation by registering over a public "action string". Elaborating on the previous example, a developer may specify the string `ACTION_VIEW` as the recipient of an implicit intent, thus enabling any image viewer registered on that string to get the message and perform the task. Implicit intents facilitate runtime binding among different applications, but are more difficult to secure [26].

Components Intents are delivered to application *components*, the essential building blocks of Android applications. There are four different types of components, serving different purposes:

- An *activity* represents a screen with a user interface. Activities are started with an intent and possibly return a result upon termination;
- A *service* runs in the background to perform long-running computations and does not provide a user interface. Services can either be started with an intent or expose a remote method invocation interface to a client upon establishment of a long-standing connection;
- A *broadcast receiver* waits for intents sent to multiple applications. Broadcast receivers typically act as forwarders of system-wide broadcast messages to specific application components;
- A *content provider* manages a shared set of persistent application data. Content providers are not accessed through intents, but through a CRUD (Create-Read-Update-Delete) interface reminiscent of SQL databases.

We refer to the first three component types as "intent-based" components. Any communication among such components can employ either explicit or implicit intents.

Protection Mechanisms The Android security model implements isolation and privilege separation on top of a simple permission system. Permissions are used both to secure (implicit) inter-component communication and to access privileged methods of the API.

Android permissions are identified by strings and can be defined by either the operating system or the applications. Permissions are granted at installation time, application-wise, and are thus shared by all the components of the same application. All permissions are assigned a protection level:

- A *normal* permission is granted to any requesting application;
- A *dangerous* permission is granted to any requesting application, provided that the user provides explicit consent;
- A *signature* permission is granted only if the requesting application is signed with the same key as the application defining the permission;
- A *signature-or-system* permission lifts the previous restriction, by also allowing a limited set of system applications to acquire the permission.

If any of the requested permissions is not assigned, the application is not installed. Permission checks may fail at runtime, whenever the granted permissions do not suffice to perform a privileged operation, leading to security exceptions.

The Android communication API offers various protection mechanisms to the different component types. In particular, all components may declare permissions which must be owned by other components requesting access; on the other hand, only by broadcasting a request one may specify permissions which a receiver must hold to handle the intent. This implies, for instance, that a programmer cannot restrict the set of possible receivers when invoking the method `startActivity` with an implicit intent: any activity registered over the string associated to the intent may be started to handle the task.

4.3 Introducing λ-`Perms`

We describe λ-`Perms`, a simple formal calculus which captures the essence of inter-component communication in Android. We detail the connections between λ-`Perms` and the Android platform in Section 4.3.2.

4.3.1 Syntax and semantics

We presuppose disjoint collections of names m, n and variables x, y, z, and use the meta-variables u, v to range over *values*, i.e., both names and variables. We denote permissions with typewriter capital letters, as in `PERMS`, and assume they form a complete lattice with partial order $\sqsubseteq$, top and bottom elements $\top$ and $\bot$ respectively, and join and meet operators $\sqcup$ and $\sqcap$.

An *expression* represents a sequential program, which runs with a given set of assigned permissions and may return a value. As part of its computation, an expression may perform function calls from a pool of *function definitions*, i.e., named expressions ready to input an argument and run. The syntax of expressions is defined in Table 4.1.

$D ::=$		*definitions*
	$\mathsf{def}\, u = \lambda(x \triangleleft \mathtt{CALL}).E$	function definition (scope of x is E)
	$D \wedge D$	conjunction
$E ::=$		*expressions*
	$D \setminus E$	evaluation
	$\overline{u}\langle v \triangleright \mathtt{RECV}\rangle$	function invocation
	$\mathsf{let}\ x = E\ \mathsf{in}\ E'$	let (scope of x is E')
	$(\nu n)\, E$	restriction (scope of n is E)
	$[\mathtt{PERMS}]\, E$	permissions assignment
	v	value

Table 4.1: Syntax of λ-Perms expressions

The expression $D \setminus E$ runs E in the pool of function definitions D. An invocation $\overline{u}\langle v \triangleright \mathtt{RECV}\rangle$ tries to call function u, supplying v as an argument; the invocation succeeds only if the callee has at least permissions RECV. A let expression $\mathsf{let}\ x = E\ \mathsf{in}\ E'$ evaluates E to a name n and then behaves as E' with x substituted by n. A restriction $(\nu n)\, E$ creates a fresh name n and then behaves as E. The expression $[\mathtt{PERMS}]\, E$ represents E running with permissions PERMS. A definition $\mathsf{def}\, u = \lambda(x \triangleleft \mathtt{CALL}).E$ introduces a function u: only callers with at least permissions CALL can invoke this function, supplying an argument for x. Multiple function definitions can be combined into a pool with the $\wedge$ operator. Function definitions, "let" and ν are binding operators for variables and names, respectively: the notions of free names *fn* and free variables *fv* arise as expected, according to the scope defined in Table 4.1.

The formal semantics of λ-Perms is given by the small-step reduction relation $E \to E'$ defined in Table 4.2. Reduction contexts $\mathcal{C}[\cdot]$ are defined as follows:

$$\mathcal{C}[\cdot] ::= \ \cdot \ \mid \ \mathsf{let}\ x = \mathcal{C}[\cdot]\ \mathsf{in}\ E \ \mid \ (\nu n)\, \mathcal{C}[\cdot] \ \mid \ D \setminus \mathcal{C}[\cdot]$$

Notice that permission assignments do not constitute a reduction context: indeed, although the syntax of expressions is liberal, such constructs are not intended to be nested.

Rule (R-CALL) implements the security "cross-check" between caller and callee, which we discussed earlier: if either the caller is not assigned permissions CALL, or the callee is not granted permissions RECV, then the function invocation fails.

(R-Call)

$$\frac{\texttt{CALL} \sqsubseteq \texttt{PERMS} \qquad \texttt{RECV} \sqsubseteq \texttt{PERMS}'}{\mathsf{def}\, n = \lambda(x \triangleleft \texttt{CALL}).[\texttt{PERMS}']\, E \setminus [\texttt{PERMS}]\, \overline{n}\langle m \triangleright \texttt{RECV}\rangle \rightarrow [\texttt{PERMS}']\, E\{m/x\}}$$

(R-Return)

$$\mathsf{let}\ x = [\texttt{PERMS}]\, n\ \mathsf{in}\ E \rightarrow E\{n/x\}$$

(R-Context)

$$\frac{E \rightarrow E'}{\mathcal{C}[E] \rightarrow \mathcal{C}[E']}$$

(R-Struct)

$$\frac{E \Rrightarrow E_1 \qquad E_1 \rightarrow E_2 \qquad E_2 \Rrightarrow E'}{E \rightarrow E'}$$

Table 4.2: Reduction semantics for λ-`Perms`

Whenever the invocation is successful, the expression runs with the permissions of the callee. The other rules are essentially standard: (R-Return) allows the execution to proceed after complete evaluation to a name n of an expression $[\texttt{PERMS}]\, E$ inside the reduction context of a let; (R-Context) states that the reduction relation is contextual; (R-Struct) closes reduction under heating, which we define as the smallest preorder closed under the rules in Table 4.3. We write $E \equiv E'$ if and only if $E \Rrightarrow E'$ and $E' \Rrightarrow E$.

We briefly discuss some aspects of the heating relation: rules (H-Extr-1) and (H-Extr-2) formalize scope extrusion, much in the same spirit as in the pi-calculus, where dynamically created names can be communicated and their scope extended to include the recipient. Rules (H-Flip-1) and (H-Flip-2) perform some housekeeping needed to export new names and functions dynamically created by a running expression: these rules are important again to enlarge the scope of these new entities. Rules (H-Comm) and (H-Assoc) are used in combination with (H-Conj) to liberally rearrange a pool of function definitions, by ignoring their order. Rule (H-Move) is needed both to perform function calls inside the reduction context of a let expression (when read from left to right) and to export new function definitions (when read from right to left). Rule (H-Distr) is borrowed from [25] and it distributes permission assignments over a let expression. Rules (H-Extr-1) and (H-Move) are adapted from the concurrent object calculus [49].

4.3.2 λ-`Perms` vs Android

Though λ-`Perms` is a small calculus, it is expressive enough to capture all the most important aspects of the Android platform of interest for our present concerns.

(H-Context)
$$\frac{E \Rrightarrow E'}{\mathcal{C}[E] \Rrightarrow \mathcal{C}[E']}$$

(H-Extr-1)
$$\frac{n \notin fn(E')}{\mathsf{let}\ x = (\nu n)\, E\ \mathsf{in}\ E' \Rrightarrow (\nu n)\, (\mathsf{let}\ x = E\ \mathsf{in}\ E')}$$

(H-Extr-2)
$$\frac{n \notin fn(D)}{D \setminus (\nu n)\, E \Rrightarrow (\nu n)\, (D \setminus E)}$$

(H-Flip-1)
$$[\mathtt{PERMS}]\, (\nu n)\, E \Rrightarrow (\nu n)\, [\mathtt{PERMS}]\, E$$

(H-Flip-2)
$$[\mathtt{PERMS}]\, (D \setminus E) \Rrightarrow D \setminus [\mathtt{PERMS}]\, E$$

(H-Comm)
$$(D_1 \wedge D_2) \setminus E \equiv (D_2 \wedge D_1) \setminus E$$

(H-Assoc)
$$(D_1 \wedge D_2) \wedge D_3 \setminus E \equiv D_1 \wedge (D_2 \wedge D_3) \setminus E$$

(H-Conj)
$$D_1 \setminus (D_2 \setminus E) \equiv (D_1 \wedge D_2) \setminus E$$

(H-Move)
$$D \setminus (\mathsf{let}\ x = E\ \mathsf{in}\ E') \equiv \mathsf{let}\ x = (D \setminus E)\ \mathsf{in}\ E'$$

(H-Distr)
$$[\mathtt{PERMS}]\ \mathsf{let}\ x = E\ \mathsf{in}\ E' \Rrightarrow \mathsf{let}\ x = [\mathtt{PERMS}]\, E\ \mathsf{in}\ [\mathtt{PERMS}]\, E'$$

Table 4.3: Heating relation for λ-`Perms`

Intents λ-`Perms` can encode both implicit and explicit intents. Communication in λ-`Perms` is non-deterministic, in that a function invocation $\overline{n}\langle m \triangleright \mathtt{RECV}\rangle$ can trigger any function definition $\mathsf{def}\, n = \lambda(x \triangleleft \mathtt{CALL}).E$ in the same scope, provided that all permission checks are satisfied. Technically, this non-determinism is enforced by the heating relation in Table 4.3, hence communication in λ-`Perms` naturally accounts for implicit intents, which represent the most interesting aspect of Android communication. Explicit intents can be recovered by univocally assigning each function definition with a distinct, unique permission: explicit communication is then encoded by requiring the callee to possess at least such permission.

Components All of Android's intent-based active component types are represented in λ-`Perms` by means of function abstractions. Activities may be started through invocations to either `startActivity` or `startActivityForResult`; in our calculus we treat the two cases uniformly, by having functions always return a result, which may simply be discarded by the caller. Services may either be started by `startService` or become the end-point of a long-running connection with a client through an invocation to `bindService`. The former behaviour is modelled directly in λ-`Perms` by a function call, while the latter is subtler and its encoding leads to some interesting findings (see below). Broadcast communication can be captured by a sequence of function invocations: this simple treatment suffices for our security analysis. Finally, there is no λ-Perms counterpart of content providers, as they are passive entities, which are not accessed through a message-passing paradigm, but through a sophisticated CRUD interface reminiscent of SQL; hence, their security verification is orthogonal to our setting.

Protection mechanisms λ-Perms is defined around a generic complete lattice of permissions. In Android this lattice is built over permission sets, with set inclusion as the underlying partial order. As to permission checking, the Android communication API only allows broadcast transmissions to be protected by permissions, namely requiring receivers to be granted specific privileges to get the message. Function invocation in λ-Perms just accounts for the more general behaviour available to broadcast transmissions, since unprotected communication can be simply encoded by specifying $\bot$ as the permission required to the callee, as in $\overline{n}\langle m \triangleright \bot\rangle$.

Binders In Android a component can invoke `bindService` to establish a connection with a service and retrieve an `IBinder` object, which transparently dispatches method calls from the client to the remote service. This behavior is captured in λ-Perms by relying on its provision for dynamic component creation. To illustrate, let D contain the following service definition:

$$D \triangleq \mathsf{def}\, s = \lambda(x \triangleleft \mathtt{C}).[\mathtt{P}]\, (\nu b)\, (\mathsf{def}\, b = \lambda(y \triangleleft \bot).[\mathtt{P}]\ \ldots \setminus b) \tag{4.1}$$

and consider the λ-Perms encoding of a component binding to service s:

$$D \setminus [\mathtt{C}]\, \mathsf{let}\ z = \overline{s}\langle n \triangleright \bot\rangle\ \mathsf{in}\ \ldots$$

Service s runs with permissions `P` and requires permissions `C` to establish a connection. When a connection is successfully established, the service returns a fresh binder b, encoded as a function granted the same permissions `P` as s. The example unveils a subtle, and potentially dangerous, behaviour of the current Android implementation of `IBinder`'s: notice in particular that the function b may be invoked with no constraint, even though binding to s was protected by permissions `C`. In Android's current implementation, in fact, the permissions checks made when binding to a service are not repeated upon method invocations over the returned `IBinder` object; we find this implementation potentially dangerous, since it is exposed to privilege escalation attacks when binders are inadvertently disclosed.

Pending intents and delegation Android introduces a form of delegation to relax the tight restrictions imposed by permissions checking. The mechanism is implemented using special objects known as *pending intents*: "by giving a `PendingIntent` to another application, you are granting it the right to perform the operation you have specified as if the other application was yourself (with the same permissions and identity)" [48]. This informal description perfectly fits the previous encoding of binders in λ-`Perms`, in that any component exposed to the binder b is allowed to invoke the corresponding function running with permissions `P`, hence pending intents can be modelled in the very same way as binders, and are exposed to the same weaknesses whenever they are improperly disclosed.

4.4 Privilege escalation, formally

Davi et al. first pointed out a conceptual weakness in the Android permission system, showing that it is vulnerable to privilege escalation attacks [29]. The problem is best illustrated with an example. Consider three applications A, B and C, each consisting of a single component. Application A is granted no permission; application B, instead, is granted permission `P`, which is needed to access C. Apparently, data and requests from A should not be able to reach C; on the other hand, since B can freely be accessed from A, then it may possibly act as a proxy between A and C (see Figure 4.1 below).

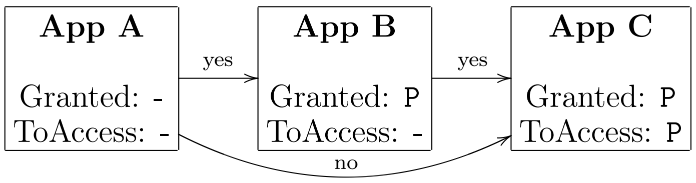

Figure 4.1: Example of privilege escalation

Defining a formal notion of safety against privilege escalation attacks is an interesting task. We start from the IPC Inspection mechanism proposed by Felt et al. to dynamically prevent privilege escalation attacks on Android [37]. The idea behind IPC Inspection is remarkably simple and strongly reminiscent of Java stack inspection: when an application receives a message from another application, a centralized runtime reference monitor lowers the privileges of the recipient to the intersection of the privileges of the two interacting applications. Since a patched Android system implementing IPC Inspection is protected against privilege escalation attacks "by design", in that function invocations may only restrict the permissions of the caller, our proposal is to consider such a system as a reference specification and state an equivalence-based notion of safety on top of it. Intuitively, an expression E is safe against privilege escalation attacks when its execution is completely oblivious of the fact that IPC Inspection is enabled or not.

Formally, we extend the syntax and we assume that each function definition occurring in a given expression is annotated with a distinct label ℓ drawn from a denumerable set $\mathcal{H}$ disjoint from the set of values. Let α range uniformly over the set of such labels and the distinguished symbol $\cdot \notin \mathcal{H}$, we let $E \xrightarrow{\alpha}_{ipc} E'$ denote the labelled reduction relation in Table 4.4. The new reduction relation $\xrightarrow{\alpha}_{ipc}$ formalizes inter-component communication in an Android system patched to support IPC Inspection. The semantics is instrumented with labels to track the dynamics of the call chains, but note that these labels do not have any import at runtime: in fact, function invocations do not mention labels at all and the semantics is still non-deterministic. We similarly label the original semantics in Table 4.2.

(R-Call-IPC)

$$\frac{\mathtt{RECV} \sqsubseteq \mathtt{PERMS}' \qquad \mathtt{CALL} \sqsubseteq \mathtt{PERMS}}{\mathsf{def}\, n^{\ell} = \lambda(x \lhd \mathtt{CALL}).[\mathtt{PERMS}']\, E \setminus [\mathtt{PERMS}]\, \overline{n}\langle m \rhd \mathtt{RECV}\rangle \xrightarrow{\ell}_{ipc} [\mathtt{PERMS} \sqcap \mathtt{PERMS}']\, E\{m/x\}}$$

(R-Return-IPC)

$$\mathsf{let}\ x = [\mathtt{PERMS}]\, n\ \mathsf{in}\ E \xrightarrow{\cdot}_{ipc} E\{n/x\}$$

(R-Context-IPC)

$$\frac{E \xrightarrow{\alpha}_{ipc} E'}{\mathcal{C}[E] \xrightarrow{\alpha}_{ipc} \mathcal{C}[E']}$$

(R-Struct-IPC)

$$\frac{E \Rrightarrow E_1 \qquad E_1 \xrightarrow{\alpha}_{ipc} E_2 \qquad E_2 \Rrightarrow E'}{E \xrightarrow{\alpha}_{ipc} E'}$$

Table 4.4: Reduction semantics for λ-Perms under IPC Inspection

We now introduce the symbol $\asymp$ to relate two expressions, with the following meaning: $E_1 \asymp E_2$ if and only if E_1 and E_2 are syntactically equal, but for their granted permissions. More precisely, we define $\asymp$ as the smallest equivalence relation on expressions closed under the rules in Table 4.5.

$$[\mathtt{P}]\, E \asymp [\mathtt{Q}]\, E \qquad \frac{E \asymp E'}{(\nu n)\, E \asymp (\nu n)\, E'} \qquad \frac{E \asymp E'}{D \setminus E \asymp D \setminus E'}$$

$$\frac{E_1 \asymp E_1' \qquad E_2 \asymp E_2'}{\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \asymp \mathsf{let}\ x = E_1'\ \mathsf{in}\ E_2'}$$

Table 4.5: Equivalence up to granted permissions

We finally have all the ingredients to adapt and recast in our setting a standard notion of simulation. The requirement $E_1 \asymp E_2$ in the next definition is needed to guarantee that the labels placed on the function definitions occurring in the two expressions are consistent (i.e., the same function bears the same label in E_1 and E_2) while dispensing from any difference in the permissions assignment introduced upon reduction (cf. (R-Call) against (R-Call-IPC)).

Definition 4.4.1 (Simulation)**.** A binary relation $\mathcal{R}$ is a *simulation* if and only if, for any pair of expressions E_1, E_2 such that $E_1 \mathcal{R} E_2$, we have $E_1 \asymp E_2$ and, whenever $E_1 \xrightarrow{\alpha} E_1'$, we have $E_2 \xrightarrow{\alpha}_{ipc} E_2'$ with $E_1' \mathcal{R} E_2'$. We say that E_1 is *simulated* by E_2 (written $E_1 \preccurlyeq E_2$) if and only if there exists a simulation $\mathcal{R}$ such that $E_1 \mathcal{R} E_2$.

Given the previous definition, our notion of safety is immediate: an expression E is safe if and only if all its possible executions are oblivious of IPC Inspection being enabled or not.

Definition 4.4.2 (Safety)**.** An expression E is *safe* against privilege escalation attacks if and only if $E \preccurlyeq E$.

Although our definition draws inspiration from IPC Inspection, it clarifies an important aspect which was never discussed before. Namely, we acknowledge that improper disclosure of some specific data, such as binders or pending intents, may lead to the development of applications which are unsafe according to Definition 4.4.2. Consider for instance the following adaptation of example (4.1):

$$D \triangleq \mathsf{def}\, s = \lambda(x \lhd \bot).[\mathtt{P}]\,(\nu b)\,(\mathsf{def}\, b = \lambda(y \lhd \bot).[\mathtt{P}]\,\overline{a}\langle y \rhd \bot\rangle \setminus b) \tag{4.2}$$

and consider an unprivileged component interacting with s:

$$(\mathsf{def}\, a = \lambda(x \lhd \mathtt{P}).[\mathtt{P}]\, E) \wedge D \setminus [\bot]\, \mathsf{let}\ z = \overline{s}\langle n \rhd \bot\rangle\ \mathsf{in}\ \overline{z}\langle n \rhd \bot\rangle$$

Service s can be freely invoked by the unprivileged component, but it returns a pending intent b, which grants access to the component a protected by permissions `P`. As such, the system does allow to escalate privileges and maliciously supply arguments to the privileged component a through the pending intent b.

Being simulation-based, our notion of safety is already a very strong property, but we target a more ambitious goal: as we discussed for RCF in Section 2.5, we desire protection despite the best efforts of an arbitrary opponent. In our model an opponent is a malicious, but unprivileged, Android application installed on the same device. Notice that the term "unprivileged" is used here in a somewhat loose sense: we are not assuming that the opponent is granted no permission at all, but rather that it is not assigned any sensitive permission beforehand (in that case, it would have no reason in escalating privileges). In a typical security analysis, one can identify all the permissions which can be acquired by the opponent (e.g., `INTERNET`) and identify the set of these permissions with $\bot$. This is feasible, since our framework is parametric with respect to a generic complete lattice of permissions.

Definition 4.4.3 (Opponent)**.** A definition O is an *opponent* if and only if each permission assignment in O is $\bot$.

Definition 4.4.4 (Robust Safety)**.** An expression E is *robustly safe* against privilege escalation attacks if and only if $O \setminus E$ is safe for all opponents O.

We conclude this section by observing that a very recent paper by Fragkaki et al. [43] proposes a formal definition of safety against privilege escalation attacks inspired by the classic notion of non-interference for information flow control. Their definition essentially demands that any call chain ending in a "high" (permission-protected) component exists in a system only if it exists in a variant of same system, where the "low" (unprivileged) components have been pruned away. We can rephrase their notion in our setting and prove that our definition of safety implies theirs.

Let $|E|_\ell$ denote the expression obtained from E by erasing all the function definitions labelled with $\ell' \neq \ell$ and which are granted permissions $\texttt{P} \sqsubset \texttt{CALL}$, where $\texttt{CALL}$ stands for the permissions required to invoke the function identified by ℓ.

Definition 4.4.5 (Alternative Safety)**.** An expression E is *safe* if and only if, for every ℓ occurring in E, we have that:

$$E \xrightarrow{\alpha_1} E_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} E_n \xrightarrow{\ell} E_{n+1}$$

implies:

$$|E|_\ell \xrightarrow{\alpha_1} E_1' \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} E_n' \xrightarrow{\ell} E_{n+1}'$$

for some $E_1', \ldots, E_{n+1}'$.

Lemma 4.4.1 (Soundness of IPC Inspection)**.** *Safety implies alternative safety.*

Proof. Let $E \preceq E$ and assume $E \xrightarrow{\alpha_1} E_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} E_n \xrightarrow{\ell} E_{n+1}$. Since $E \preceq E$, we know that $E \xrightarrow{\alpha_1}_{ipc} E_1' \xrightarrow{\alpha_2}_{ipc} \ldots \xrightarrow{\alpha_n}_{ipc} E_n' \xrightarrow{\ell}_{ipc} E_{n+1}'$ for some $E_1', \ldots, E_{n+1}'$ such that $E_1 \asymp E_1', \ldots, E_{n+1} \asymp E_{n+1}'$. By definition of the semantics $\xrightarrow{\alpha}_{ipc}$, we know that all the functions invoked in the call chain identified by $\alpha_1, \ldots, \alpha_n$ must be granted at least the permissions $\texttt{CALL}$ needed to invoke ℓ. Hence, such function definitions are present also in $|E|_\ell$ and we can mimic the very same trace there. $\square$

Thus, we can formally confirm that the IPC Inspection mechanism enforces a reasonable semantic security property and justify our choice of taking it as the building block for our notion of safety. With respect to the alternative definition, our notion is somewhat less intuitive, but it has the important advantage of enabling a powerful form of coinductive reasoning, which is central to proving our main result (Theorem 4.5.3 below).

A still open question is if the alternative notion of safety is actually equivalent to ours. We notice that for non-deterministic transition systems (bi)simulation-based equivalences are typically finer than trace equivalences, but at the time of writing we were not able to identify a counterexample showing that the alternative notion of safety is weaker than our definition in the present setting.

4.5 Preventing privilege escalation, by typing

We present a static type-and-effect system which allows us to enforce robust protection against privilege escalation attacks. Designing a sound type discipline is subtle, mainly due to the presence of sensitive data like binders and pending intents, which the opponent may actively try to get under its control by deceiving well-typed components.

Types and typing environments We consider a minimal syntax for types, given below.

$$\tau ::= \ \mathsf{Un} \ \mid \mathsf{Fun}(\mathtt{CALL}, \tau \rightarrow \tau')^{\mathtt{SECR}}$$

Type Un is the base type, which is used both as a building block for function types and to encompass all the data which are under the control of the opponent. Types of the form $\mathsf{Fun}(\mathtt{CALL}, \tau \rightarrow \tau')^{\mathtt{SECR}}$ are inhabited by functions which input arguments of type τ and return results of type τ'. Functions with this type can be invoked only by callers which are granted at least permissions $\mathtt{CALL}$, and should only be disclosed to components running with at least permissions $\mathtt{SECR}$.

We define the *secrecy level* of a type τ, written $\mathcal{L}(\tau)$, as expected, by having $\mathcal{L}(\mathsf{Un}) = \bot$ and $\mathcal{L}(\mathsf{Fun}(\mathtt{CALL}, \tau \rightarrow \tau')^{\mathtt{SECR}}) = \mathtt{SECR}$. A typing *environment* Γ is a finite map from values to types. The *domain* of a typing environment Γ, written $dom(\Gamma)$, is the set of the values on which Γ is defined.

Typing values The typing rules for values are simple, and given below.

$$\text{(T-Proj)} \quad \frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau} \qquad\qquad \text{(T-Pub)} \quad \frac{\Gamma \vdash v : \tau \qquad \mathcal{L}(\tau) = \bot}{\Gamma \vdash v : \mathsf{Un}}$$

(T-Proj) is standard, while (T-Pub) makes it possible to treat all public data as "untyped", since they may possibly be disclosed to the opponent. We discuss the type rules for opponent code in the next section.

Typing expressions The typing rules for expressions are in Table 4.6. The main judgement $\Gamma \vdash_{\mathtt{PERMS}} E : \tau \blacktriangleright \mathtt{PERMS}'$ is read as "expression E, running with permissions $\mathtt{PERMS}$, has type τ in Γ and exercises at most permissions $\mathtt{PERMS}'$ throughout its execution". We also define an auxiliary judgement $\Gamma \vdash D$ to be read as "definition D is well-formed in Γ". The two judgement forms are mutually dependent.

We first note that our effect system discriminates between *granted* permissions and *exercised* permissions. For instance, the expression:

$$\mathsf{def}\ a = \lambda(x \triangleleft \bot).[\mathtt{P}]\, \overline{b}\langle n \triangleright \bot\rangle \setminus ...$$

(T-Def)
$$\frac{\begin{array}{c}\Gamma \vdash u : \mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}} \\ \Gamma, x : \tau \vdash_{\top} E : \tau' \blacktriangleright \mathtt{PERMS}' \qquad \mathtt{PERMS}' \sqsubseteq \mathtt{CALL} \sqcup \mathtt{SECR} \\ \mathtt{CALL} \sqcup \mathtt{SECR} = \bot \Rightarrow \Gamma, x : \mathsf{Un} \vdash_{\top} E : \mathsf{Un} \blacktriangleright \bot \qquad x \notin \mathit{dom}(\Gamma)\end{array}}{\Gamma \vdash \mathsf{def}\, u = \lambda(x \triangleleft \mathtt{CALL}).E}$$

(T-Conj)
$$\frac{\Gamma \vdash D_1 \qquad \Gamma \vdash D_2}{\Gamma \vdash D_1 \wedge D_2}$$

(T-Eval)
$$\frac{\Gamma \vdash D \qquad \Gamma \vdash_{\mathtt{PERMS}} E : \tau \blacktriangleright \mathtt{PERMS}'}{\Gamma \vdash_{\mathtt{PERMS}} D \setminus E : \tau \blacktriangleright \mathtt{PERMS}'}$$

(T-Call)
$$\frac{\begin{array}{c}\Gamma \vdash u : \mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}} \qquad \Gamma \vdash v : \tau \\ \bot \sqsubset \mathtt{RECV} \sqcup \mathtt{SECR} \qquad \mathtt{CALL} \sqcup \mathtt{SECR} \sqsubseteq \mathtt{PERMS}\end{array}}{\Gamma \vdash_{\mathtt{PERMS}} \overline{u}\langle v \triangleright \mathtt{RECV}\rangle : \tau' \blacktriangleright \mathtt{CALL} \sqcup \mathtt{SECR}}$$

(T-Val)
$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash_{\mathtt{PERMS}} v : \tau \blacktriangleright \bot}$$

(T-Fail)
$$\frac{\begin{array}{c}\Gamma \vdash u : \mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}} \qquad \Gamma \vdash v : \tau'' \\ \mathtt{RECV} \sqcup \mathtt{SECR} = \bot \Rightarrow \mathcal{L}(\tau'') = \bot \qquad \mathtt{CALL} \not\sqsubseteq \mathtt{PERMS}\end{array}}{\Gamma \vdash_{\mathtt{PERMS}} \overline{u}\langle v \triangleright \mathtt{RECV}\rangle : \mathsf{Un} \blacktriangleright \mathtt{PERMS}}$$

(T-Perms)
$$\frac{\begin{array}{c}\Gamma \vdash_{\mathtt{PERMS}'} E : \tau \blacktriangleright \mathtt{PERMS}'' \\ \mathtt{PERMS}' \sqsubseteq \mathtt{PERMS}\end{array}}{\Gamma \vdash_{\mathtt{PERMS}} [\mathtt{PERMS}']\, E : \tau \blacktriangleright \mathtt{PERMS}''}$$

(T-Let)
$$\frac{\begin{array}{c}\Gamma \vdash_{\mathtt{PERMS}} E : \tau \blacktriangleright \mathtt{PERMS}' \\ \Gamma, x : \tau \vdash_{\mathtt{PERMS}} E' : \tau' \blacktriangleright \mathtt{PERMS}'' \qquad x \notin \mathit{dom}(\Gamma)\end{array}}{\Gamma \vdash_{\mathtt{PERMS}} \mathsf{let}\ x = E\ \mathsf{in}\ E' : \tau' \blacktriangleright \mathtt{PERMS}' \sqcup \mathtt{PERMS}''}$$

(T-Restr)
$$\frac{\begin{array}{c}\Gamma, n : \tau \vdash_{\mathtt{PERMS}} E : \tau' \blacktriangleright \mathtt{PERMS}' \\ n \notin \mathit{dom}(\Gamma)\end{array}}{\Gamma \vdash_{\mathtt{PERMS}} (\nu n)\, E : \tau' \blacktriangleright \mathtt{PERMS}'}$$

(T-Def-Un)
$$\frac{\begin{array}{c}\Gamma \vdash u : \mathsf{Un} \\ \Gamma, x : \mathsf{Un} \vdash_{\bot} E : \mathsf{Un} \blacktriangleright \bot \\ x \notin \mathit{dom}(\Gamma)\end{array}}{\Gamma \vdash \mathsf{def}\, u = \lambda(x \triangleleft \mathtt{CALL}).E}$$

(T-Call-Un)
$$\frac{\Gamma \vdash u : \mathsf{Un} \qquad \Gamma \vdash v : \mathsf{Un}}{\Gamma \vdash_{\bot} \overline{u}\langle v \triangleright \mathtt{RECV}\rangle : \mathsf{Un} \blacktriangleright \bot}$$

Table 4.6: Typing rules for λ-`Perms`

could either be well-typed or not, even though the function a is publicly available, but runs with strong permissions $\texttt{P} \sqsupset \bot$. The crux here is if the permissions $\texttt{P}$ are indeed necessary to perform the invocation to b or not. We take advantage of the information tracked by our effect system in a number of type rules, as well as to perform additional helpful checks in our tool (see Section 4.6). Below, we comment on the most interesting (aspects of the) rules.

We consider rule (T-Def) first. The third condition is central to enforce protection against privilege escalation. Namely, invoking a function of type $\mathsf{Fun}(\texttt{CALL}, \tau \rightarrow \tau')^{\texttt{SECR}}$ requires both permissions $\texttt{CALL}$, to pass the security runtime checks, and permissions $\texttt{SECR}$, to learn the name of the function; this implies that $\texttt{CALL} \sqcup \texttt{SECR}$ is a lower bound for the permissions granted to any caller of the function. Therefore, if the permissions exercised by the function itself are bounded above by $\texttt{CALL} \sqcup \texttt{SECR}$, no caller can escalate privileges upon invocation. As a practical remark, recall that in Android both binders and pending intents enable indiscriminate access to a given application component c upon disclosure. Hence, our type system forces to assign to such values a secrecy level which is at least as high as the permissions exercised by the component c. For instance, in example (4.2), we would give b a type of the form $\mathsf{Fun}(\bot, \tau_b \rightarrow \tau_b')^{\texttt{P}}$ for some τ_b, τ_b'.

Continuing with rule (T-Def), the fourth condition is needed to account for interactions with the opponent. Since a function of type $\mathsf{Fun}(\bot, \tau \rightarrow \tau')^{\bot}$ is public and can be invoked by anyone, the body of such function must be type-checked also under the assumption that the input parameter is provided by the opponent (with type Un). Of course, in such case no privilege must be exercised by the function. A similar treatment is enforced by security type systems including cryptography to handle asymmetric decryption, since messages encrypted under a public key may actually come from the opponent [38, 4].

We now focus on rule (T-Call). Its first two conditions are standard, while the third one is needed to rule out as ill-typed the invocation $\overline{u}\langle v \rhd \bot\rangle$ when u is public. This is a very subtle case, since function invocation is non-deterministic in λ-$\texttt{Perms}$, hence the previous call, which does not constrain at all the choice of the callee, may run either a function defined by the opponent or a piece of trusted code. In the first case we should consider Un as the return type, while in the second case we should expect some value of type τ'. It turns out that both choices are unsound: the first one could break the secrecy of the return value upon interaction with trusted code; the second one would give the strong type τ' to some tainted data returned by the opponent. The implication for the Android platform is that any call to `startActivityForResult` or to `bindService` should employ explicit intents to be deemed as well-typed.

The last condition of rule (T-Call) is specifically designed to prevent privilege escalation attacks. Indeed, recall that a function of type $\mathsf{Fun}(\texttt{CALL}, \tau \rightarrow \tau')^{\texttt{SECR}}$ can exercise at most privileges $\texttt{CALL} \sqcup \texttt{SECR}$ by rule (T-Def), hence it can be safely invoked only by a caller granted with at least permissions $\texttt{PERMS} \sqsupseteq \texttt{CALL} \sqcup \texttt{SECR}$. This interplay between rules (T-Call) and (T-Def) implements a rely-guarantee mechanism common to the modular analysis performed by most type systems.

The opponent counterparts for rules (T-Def) and (T-Call) are rules (T-Def-Un) and (T-Call-Un) respectively. By using these rules, the opponent can define arbitrary new functions and invoke existing ones, completely disregarding the restrictions enforced by typing. These rules are needed only for technical reasons, namely allowing us to prove Theorem 4.5.3 below; as such, they are not included in our implementation.

Finally, we discuss rule (T-Fail). This rule is tricky and it is not strictly needed for soundness, but just to make type-checking more precise. To illustrate, consider the invocation $\overline{u}\langle v \triangleright \mathtt{RECV}\rangle$ performed by a caller endowed with permissions $\mathtt{PERMS}$ and assume that u has type $\mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}}$. We can distinguish two cases: either u is defined by trusted code through rule (T-Def), or u is defined by the opponent using rule (T-Def-Un). In the first case, the information $\mathtt{CALL}$ annotated on the function type is consistent with the runtime permission enforcement, thus, since $\mathtt{CALL} \not\sqsubseteq \mathtt{PERMS}$, we are guaranteed that the invocation will actually fail at runtime and we can give an arbitrary type τ'' to the argument v. Otherwise, suppose that u was defined by the opponent: in this case the invocation might actually take place, since the opponent can disregard the type of u. Anyway, if the invocation happens, we are guaranteed that $\mathtt{RECV} \sqcup \mathtt{SECR} \sqsubseteq \bot$, since the opponent has no privileges and learns only public data; we must then enforce the condition $\mathcal{L}(\tau'') \sqsubseteq \bot$ to protect the secrecy of the argument v. Note that, due to such a possible interaction with the opponent, the exercised permissions are conservatively assumed as $\mathtt{PERMS}$, i.e., all the permissions granted to the caller.

We conclude the description of the type system with an important remark on expressiveness. Some of the constraints imposed by our typing rules are rather restrictive for practical use, but are central to enforcing the conditions of Definition 4.4.2 and its robust variant. Our implementation, however, features a number of escape hatches based on Java annotations to keep programming practical, much in spirit of the declassification/endorsement constructs customary to the information-flow literature [62]. We discuss this point further in Section 4.6.3.

Formal results We can prove that the previous type discipline enforces the expected security properties. The safety result below follows by a "simulation-aware" variant of a standard Subject Reduction theorem for our type system, which captures the step-by-step relationships between the standard semantics and our reference semantics based on IPC Inspection. The proof relies on a co-inductive argument enabled by the Subject Reduction theorem, full details can be found in Chapter 5.

Theorem 4.5.1 (Type Safety). *If $\Gamma \vdash_{\top} E : \tau \blacktriangleright \mathtt{P}$, then $E \preccurlyeq E$.*

The next result states that our type system does not constrain the opponent.

Lemma 4.5.2 (Opponent Typability). *Let O be an opponent and let $\Gamma \vdash u : \mathsf{Un}$ for all $u \in fnfv(O)$, then $\Gamma \vdash O$.*

By combining the two previous results, we can prove our main theorem.

Theorem 4.5.3 (Robust Safety). *Let $\mathcal{L}(\tau) = \bot$ for every u such that $\Gamma(u) = \tau$. If $\Gamma \vdash_{\top} E : \tau \blacktriangleright \mathtt{P}$, then E is robustly safe against privilege escalation attacks.*

4.6 Implementation

Our implementation is a tool (`Lintent`) designed as a plug-in for Android `Lint`, the official static analysis utility distributed within the Android Development Tools (ADT). The design of the tool was conducted in collaboration with Alvise Spanò and many challenges and solutions have been discussed together, but the actual implementation is entirely due to him. A more thorough description of `Lintent` can be found in Alvise's thesis [70].

`Lintent` analyzes Java source code rather than bytecode, since it has been developed within a larger research project aimed at devising type-based verification techniques for Android applications. In principle, the same analysis could be performed on the bytecode, though reasoning about types at the bytecode level is arguably more demanding than at source level [45].

The main highlights of `Lintent` may be summarized as follows.

ADT Lint integration Android `Lint` is a very useful ADT component, as it can detect a wide range of anomalies and defects within the source code and related meta-data (manifest file, resource files, etc.) that the Java compiler alone would not be able to spot out. `Lint` is very popular within the development community, therefore deploying our tool as a `Lint` plug-in appears to be the natural choice to ease a wide adoption.

Security verification The Java compiler is completely oblivious of the Android permission system, since all permission information is encoded in terms of string literals used within the Java code and declared in the manifest. At the time of writing, even Android `Lint` does not perform any static check on permissions usage, thus leaving developers exposed, for instance, to run-time failures once the Android operating system detects a permission violation on some component interaction. `Lintent` performs a number of static checks over permissions usage, analyzing the application source code and the manifest permission declarations, and eventually warning the developer in case of potential attack surfaces for privilege escalation scenarios. As a byproduct of its analysis, `Lintent` is able to detect over-privileged or under-privileged applications, and suggest fixes.

Intent and component type reconstruction The typing of intents and component supported by the Java compiler is rather loose and uninformative: in fact, the Java type system does not keep track of any type information about either the contents of Intent objects, or the data a component sends and expects to receive. This seriously hinders any form of type-based analysis, including the one discussed in this chapter, and makes Android programming very error-prone. `Lintent` infers and records the types of data injected into and extracted out of intents, while tracking the flow of inter-component message passing for reconstructing the incoming requests and outgoing results of each component. This is needed to prevent improper disclosure of binders or pending intents, but it proves helpful also to detect common programming errors related to misuse of intents [58].

4.6.1 Architecture

The `Lintent` architecture is described in Figure 4.2 below.

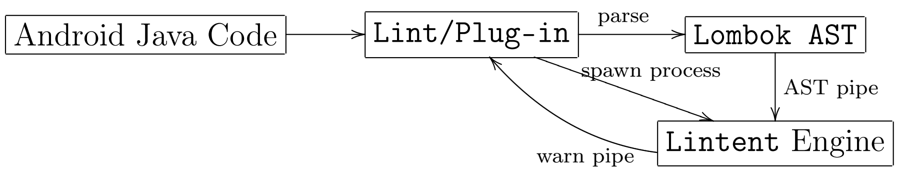

Figure 4.2: `Lintent` architecture

As anticipated, the tool is a `Lint` plug-in acting as a front-end for an engine program running as a separate process. The plug-in is written in Java and takes advantage of the built-in Java parser offered by `Lint`, which produces an Abstract Syntax Tree (AST) based on `Lombok JavaC AST` [65]. Once parsing ends successfully, the engine process is spawned and starts receiving data from a pipe formerly created by the plug-in itself for interprocess communication. Our plug-in AST visitor simply serializes the program tree through the pipe and then waits for feedback from the engine process, hanging on a second pipe aimed at receiving warnings and messages to be eventually shown as issues by the `Lint` UI. The engine program is written in F# and does the real job: after deserializing the input program tree acquired from the AST pipe, it creates its own custom representation of the AST and performs the analysis.

The first phase consists in reconstructing the types of intents and components by means of a hybrid type-inference/partial-evaluation algorithm; the second pass eventually checks permissions usage and validates security-related properties of the input program. Throughout the analysis, the engine communicates back with the `Lint` plug-in through the warn pipe, feeding back any issue worth to be prompted to the user.

4.6.2 Challenges

Analyzing Android applications is a complex and demanding activity, which involves a number of non-trivial inter-related tasks.

Detecting API patterns Implementing the rules from the abstract type system for λ-Perms requires a preliminary analysis to detect the corresponding patterns in the Android source code. The analysis is far from trivial, given the complexity of the Android communication API, which offers various different patterns to implement inter-component communication. For example, the developer guide describes at least three different ways to implement bound services, with different degrees of complexity, and a local inspection of the instructions in isolation does not suffice to reconstruct enough information to support verification. Partial evaluation techniques combined with type inference are needed where syntactic pattern matching of code templates would be too naive.

Delocalized information Permissions in Android are meta-information which are not included in Java sources, but in the application Manifest file. This is an XML file containing, among other information, the permissions each application component requires for being accessed and what permissions are requested by the application itself. Several Android API calls require non-empty permission sets and must be detected and tracked by our tool. `Lintent` retrieves a set of mappings between API method signatures and permissions from a set of external files[1], which are thus updatable with no need to rebuild the tool. All this information is needed to implement our effect system and is central to type-checking.

Type reconstruction Arguably the hardest challenge arising during the implementation is related to a number of “untyped” programming conventions which are enabled by the current Android API. Consider, for instance, a simple scenario of intent usage with multiple data types:

```
class MySenderActivity extends Activity {
   static class MySer implements Serializable { ... }

   void mySenderMethod() {
      Intent i = new Intent(this, TargetActivity.class);
      i.putExtra("k1", 3);
      i.putExtra("k2", "some_string");
      i.putExtra("k3", new MySer());
      startActivityForResult(i, 0);
   }
}
```

Since the `putExtra` method is overloaded to different types, the type of the second argument of each call must be reconstructed in order to keep track of the actual type of the value bound to each key. On the recipient side, intent “extras” are retrieved by freely accessing the intent as if it was a dictionary, so the receiver may actually retrieve data of unexpected type and fail at runtime, or disregard altogether some fields provided by the sender.

[1]Currently such permission map files are those distributed along with Stowaway [35].

```
class MyRecipientActivity extends Activity {
    static class WrongSer implements Serializable { ... }

    void onCreate(Bundle savedInstanceState) {
        Intent i = getIntent()
        // run-time type error: k1 was an int!
        String k1 = i.getStringExtra("k1");
        // dynamic cast fails!
        WrongSer o = (WrongSer)i.getSerializableExtra("k3");
        // forgets to extract "k2": might be unwanted!
    }
}
```

The example highlights a total lack of static control over standard intents manipulation operations: with these premises, no type-based analysis can be soundly performed. For this reason, intents are treated in `Lintent` as record types of the form $\{k_1 : T_1, \ldots, k_n : T_n\}$, where k_i is a string constant and T_i is a Java type. This enforces a much stronger discipline on data passing between components, i.e., on the injection and extraction of "extras" into and from intents. Notably, the same type reconstruction applies to objects of type `Bundle` as well, and `Bundle` objects possibly put within Intents or other `Bundle`'s are recursively typed as sub-records. Our treatment is consistent with our type system, in that a function type $\mathsf{Fun}(\mathtt{CALL}, \tau \rightarrow \tau')^{\mathtt{SECR}}$ constrains the caller in providing an argument (i.e., an intent) of type τ and the callee in returning a result of type τ'. Enforcing the same discipline for Android applications is crucial to protect the secrecy of binders and pending intents. As a byproduct of this analysis, our tool is able to warn the user in case of ill-typed or dangerous manipulations of the intent.

Partial evaluation Recall from the previous discussion that every data an user puts into an intent must be bound to a key, hence an intent object can be thought as a dictionary of the form $\{k_1 \mapsto v_1, \ldots, k_n \mapsto v_n\}$. Unfortunately, the dictionary keys are run-time string objects and therefore plain expressions in Java – they are not first-class language identifiers. Whether they happen to be string literals or complex method calls computing a string object is irrelevant: in any case they belong to the run-time world. The very same problem arises for result codes and Intent constructor invocation: both the sender component and the recipient class object supplied as arguments could be results of computations, and the same holds true for action strings in case of implicit intent construction. Partial evaluation during type-checking is required for reconstructing the intent record type labels described above.

Interaction with third party libraries Typically applications rely on external libraries offering a number of services to the programmer. From the point of view of Java code, such libraries are collections of compiled classes linked into one or more `jar` files: their source code is therefore not available at analysis time. Import declarations on top of compilation units simply carry information on package names and class paths, but do not specify class member signatures or other details. Type resolution is a tricky task for a tool that does not have the same information the compiler is given by command line arguments, therefore types that are inferred as external must be treated in some special way: access to `jar` files must be granted to `Lintent` to let it inspect the contents of imported packages and classes.

4.6.3 Java annotations support

We rely on Java annotations to provide a number of escape hatches from the tight discipline imposed by our type rules. Several privileged components intentionally expose functionalities, hence we define annotations of the form `@priv{endorse="P"}` to mark methods such as `onCreate()` with a set of permissions `P` which can be dispensed by the type-checker. Namely, if the method exercises the permissions set `Q`, its containing component is deemed as well-typed if it is protected with at least permissions `Q` $\setminus$ `P`. A similar treatment is implemented for pending intents through the usage of the annotation `@priv{declassify="P"}`, which allows to reduce the secrecy level of these objects computed by our type-checker, and enables a more controlled form of delegation.

4.6.4 Limitations and extensions

At the moment the tool supports only activities and started services, while support for bound services is still under heavy development and in a very preliminary stage. We plan to identify calls to API methods as `checkCallingPermissions()` to make our static analysis more precise. We are also investigating the possibility of developing a frontend to a decompiler as `smali` [2] or `ded` [33] to support the analysis of third-party applications.

4.7 Related work

There exists a huge literature on Android application security, as recently reported in an interesting survey by Enck [31].

Android permissions The deficiencies of the Android permission system with respect to privilege escalation attacks were first pointed out by Davi et al. [29]. The paper presents a proof of concept attack, but does not discuss any possible solution to the problem. Felt et al. instead propose a runtime mechanism called IPC inspection to provide protection against privilege escalation attacks on Android [37]. The solution is reminiscent of Java stack inspection and it inspired our definition of safety, as we discussed in Section 4.4. We find the implementation design very competent, but we also notice that IPC inspection may induce substantial performance overhead, since it requires keeping track of different application instances to make the protection mechanism precise, and avoid impacting heavily on the user's experience. In a more recent work, Bugiel et al. describe a fairly sophisticated runtime framework for enforcing protection against privilege escalation attacks on Android [18]. Notably, their solution comprises countermeasures also against colluding applications, which maliciously collaborate to escalate privileges, an aspect which is neglected by both IPC inspection and our type system. Providing such guarantees, however, requires a centralized solution built over low-level operating system mechanisms. We aim at being complementary to such proposal: enforcing runtime protection is fundamental against malicious applications which reach the market, while static analysis techniques can be helpful for well-meaning developers who desire to validate, and possibly certify, their code. Finally, Felt et al. propose Stowaway, a static analysis tool for detecting overprivilege in Android applications [36]. In our implementation we take advantage of their permission map, which relates API method calls to their required permissions.

Android communication The threats related to the Android message-passing system were first studied by Chin et al. [26]. Their paper provides an interesting overview of the intent-based attack surfaces and discusses guidelines for secure communication. The authors provide also a tool, ComDroid, which is able to detect potential vulnerabilities in the usage of intents. However, the paper does not provide any formal guarantee about the effectiveness of the proposed secure communication guidelines; in our work, instead, we reason about intents usage in a formal calculus, hence we are able to confirm many of their findings as sound programming practices. ComDroid does not address the problem of detecting privilege escalation attacks. The robustness of inter-component communication in Android has been studied also by Maji et al. through the usage of fuzzy testing techniques, exposing some interesting findings [58]. Their empirical methodology, however, does not provide a clear understanding of the correct programming patterns for communication.

Formal models λ-`Perms` is partially inspired by a core formal language proposed by Chaudhuri [25]. With respect to such formalism, however, λ-`Perms` provides a more thorough treatment of a number of Android peculiarities. First, it provides support for implicit communication and runtime registration of new components over action strings, which are arguably among the most interesting features from a security point of view. Second, it introduces a scoping construct, which is useful to model both service binding and pending intents; in general, a restriction operator in the style of process algebras typically proves useful for formal security reasoning. In later work, Fuchs et al. build on the calculus proposed by Chaudhuri to implement SCanDroid, a provably sound static checker of information-flow properties of Android applications [44]. Another work by Fragkaki et al. discusses a number of enhancements over the Android permission system and validates their effectiveness in an abstract model [43]. Most notably, as we mentioned, the paper proposes a formal definition of protection against privilege escalation attacks inspired to the classic notion of non-interference. The paper also discusses some issues related to controlled delegation, but it does it independently from privilege escalation. The focus of the work is on runtime protection mechanisms. Shin et al. introduce a mechanized model of the Android permission system and validate some expected security properties using Coq [69]. Language support for privilege-based software systems has been studied by Jagadeesan et al. [54] and Braghin et al. [17].

Chapter 5

Proofs of Chapter 4

We detail a full proof of soundness for the type-and-effect system of Section 4.5. In the next results we unfold the (R-CONTEXT)/(H-CONTEXT) rule from Table 4.2/ 4.3 into a number of different reduction/heating rules, one for each possible context.

5.1 Basic results

Notation 5.1.1. We adopt the following notational conventions:

(i) We often write $\Gamma \vdash E : \tau \blacktriangleright \mathtt{P}$ when $\Gamma \vdash_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{P}$ for some $\mathtt{Q}$.

(ii) We write $\Gamma \vdash^{\xi}_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{P}$ if ξ is a type derivation ending with $\Gamma \vdash_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{P}$.

(iii) We write $\Gamma \vdash_{\mathtt{Q}} \mathcal{J}$ to stand for any of the following judgements:

- $\Gamma \vdash u : \tau$ for some u and τ
- $\Gamma \vdash D$ for some D
- $\Gamma \vdash_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{P}$ for some E, τ and $\mathtt{P}$.

Proposition 5.1.1 (Uniqueness of Function Types). *If $\Gamma \vdash u : \mathit{Fun}(\mathtt{CALL}, \tau_1 \to \tau_2)^{\mathtt{SECR}}$ and $\Gamma \vdash u : \mathit{Fun}(\mathtt{CALL}', \tau_1' \to \tau_2')^{\mathtt{SECR}'}$, then $\mathtt{CALL} = \mathtt{CALL}'$, $\tau_1 = \tau_1'$, $\tau_2 = \tau_2'$ and $\mathtt{SECR} = \mathtt{SECR}'$.*

Proof. Immediate by inspection of the type rules, since the only rule which can derive function types is (T-PROJ) and Γ is a map from values to types. □

Proposition 5.1.2 (Soundness of Secrecy Levels). *If $\Gamma \vdash u : \tau$ and $\Gamma \vdash u : \tau'$, then $\mathcal{L}(\tau) = \mathcal{L}(\tau')$.*

Proof. By induction on the sum of the depth of the derivations of $\Gamma \vdash u : \tau$ and $\Gamma \vdash u : \tau'$. The only interesting case is when $\Gamma \vdash u : \tau$ was derived by (T-PROJ) and $\Gamma \vdash u : \tau'$ was derived by (T-PUB), or vice-versa. Without loss of generality, consider the first possibility: in this case we know that $\Gamma \vdash u : \tau$ by the premise $\Gamma(u) = \tau$ and $\Gamma \vdash u : \tau'$ with $\tau' = \mathsf{Un}$ by the premise $\Gamma \vdash u : \tau''$ for some τ'' such that $\mathcal{L}(\tau'') = \bot$. By inductive hypothesis we then have $\mathcal{L}(\tau) = \mathcal{L}(\tau'') = \bot$. We conclude by noting that $\mathcal{L}(\tau') = \mathcal{L}(\mathsf{Un}) = \bot = \mathcal{L}(\tau)$. □

Lemma 5.1.3 (Weakening). *If $\Gamma \vdash_{\mathtt{Q}} \mathcal{J}$ and $u \notin dom(\Gamma)$, then $\Gamma, u : \tau \vdash_{\mathtt{Q}} \mathcal{J}$. (Moreover, the effects computed throughout the entire type derivation do not change.)*

Proof. By a standard induction on the derivation of $\Gamma \vdash_{\mathtt{Q}} \mathcal{J}$. □

Lemma 5.1.4 (Substitution). *Let $\Gamma, x : \tau \vdash_{\mathtt{Q}} \mathcal{J}$ with $x \notin dom(\Gamma)$. If $\Gamma \vdash n : \tau$, then $\Gamma \vdash_{\mathtt{Q}} \mathcal{J}\{n/x\}$. (Moreover, the effects computed throughout the entire type derivation do not change.)*

Proof. By a standard induction on the derivation of $\Gamma, x : \tau \vdash_{\mathtt{Q}} \mathcal{J}$. □

Lemma 5.1.5 (Heating Preserves Typing). *If $\Gamma \vdash_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{P}$ and $E \Rrightarrow E'$, then $\Gamma \vdash_{\mathtt{Q}} E' : \tau \blacktriangleright \mathtt{P}$.*

Proof. By induction on the derivation of $E \Rrightarrow E'$. The reflexivity case is trivial and the transitivity case immediately follows by inductive hypothesis, so we focus on the remaining rules:

Case (H-Eval): let $D \setminus E \Rrightarrow D \setminus E'$ by the premise $E \Rrightarrow E'$. Since $\Gamma \vdash_{\mathtt{Q}} D \setminus E : \tau \blacktriangleright \mathtt{P}$, we have $\Gamma \vdash D$ and $\Gamma \vdash_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{P}$ by (T-Eval). By inductive hypothesis $\Gamma \vdash_{\mathtt{Q}} E' : \tau \blacktriangleright \mathtt{P}$, hence $\Gamma \vdash_{\mathtt{Q}} D \setminus E' : \tau \blacktriangleright \mathtt{P}$ by (T-Eval);

Case (H-Let): assume $\mathsf{let}\ x = E\ \mathsf{in}\ E'' \Rrightarrow \mathsf{let}\ x = E'\ \mathsf{in}\ E''$ by the premise $E \Rrightarrow E'$. Since $\Gamma \vdash_{\mathtt{R}} \mathsf{let}\ x = E\ \mathsf{in}\ E'' : \tau \blacktriangleright \mathtt{PERMS}$, we have $\Gamma \vdash_{\mathtt{R}} E : \tau' \blacktriangleright \mathtt{P}$ and $\Gamma, x : \tau' \vdash_{\mathtt{R}} E'' : \tau \blacktriangleright \mathtt{Q}$ with $\mathtt{P} \sqcup \mathtt{Q} = \mathtt{PERMS}$ by (T-Let). By inductive hypothesis $\Gamma \vdash_{\mathtt{R}} E' : \tau' \blacktriangleright \mathtt{P}$, hence $\Gamma \vdash_{\mathtt{R}} \mathsf{let}\ x = E'\ \mathsf{in}\ E'' : \tau \blacktriangleright \mathtt{PERMS}$ by (T-Let);

Case (H-Restr): let $(\nu n)\, E \Rrightarrow (\nu n)\, E'$ by the premise $E \Rrightarrow E'$. Since $\Gamma \vdash_{\mathtt{Q}} (\nu n)\, E : \tau' \blacktriangleright \mathtt{PERMS}$, we have $\Gamma, n : \tau \vdash_{\mathtt{Q}} E : \tau' \blacktriangleright \mathtt{PERMS}$ by (T-Restr). By inductive hypothesis $\Gamma, n : \tau \vdash_{\mathtt{Q}} E' : \tau' \blacktriangleright \mathtt{PERMS}$, hence $\Gamma \vdash_{\mathtt{Q}} (\nu n)\, E' : \tau' \blacktriangleright \mathtt{PERMS}$ by (T-Restr).

Case (H-Extr-1): assume $\mathsf{let}\ x = (\nu n)\, E_1\ \mathsf{in}\ E_2 \Rrightarrow (\nu n)\, (\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2)$ with $n \notin fn(E_2)$. Since $\Gamma \vdash_{\mathtt{R}} \mathsf{let}\ x = (\nu n)\, E_1\ \mathsf{in}\ E_2 : \tau_2 \blacktriangleright \mathtt{PERMS}$, we have $\Gamma \vdash_{\mathtt{R}} (\nu n)\, E_1 : \tau_1 \blacktriangleright \mathtt{P}$ and $\Gamma, x : \tau_1 \vdash_{\mathtt{R}} E_2 : \tau_2 \blacktriangleright \mathtt{Q}$ with $\mathtt{P} \sqcup \mathtt{Q} = \mathtt{PERMS}$ and $x \notin dom(\Gamma)$ by (T-Let). The former judgement can be derived only by (T-Restr), hence we have $\Gamma, n : \tau \vdash_{\mathtt{R}} E_1 : \tau_1 \blacktriangleright \mathtt{P}$ with $n \notin dom(\Gamma)$. Now we apply Lemma 5.1.3 (Weakening) to derive $\Gamma, n : \tau, x : \tau_1 \vdash_{\mathtt{R}} E_2 : \tau_2 \blacktriangleright \mathtt{Q}$ from $\Gamma, x : \tau_1 \vdash_{\mathtt{R}} E_2 : \tau_2 \blacktriangleright \mathtt{Q}$, hence we have $\Gamma, n : \tau \vdash_{\mathtt{R}} \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \tau_2 \blacktriangleright \mathtt{PERMS}$ by (T-Let) and we conclude $\Gamma \vdash_{\mathtt{R}} (\nu n)\, (\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2) : \tau_2 \blacktriangleright \mathtt{PERMS}$ by (T-Restr);

Case (H-Extr-2): let $D \setminus (\nu n)\, E \Rrightarrow (\nu n)\, (D \setminus E)$ with $n \notin \mathit{fn}(D)$. Given that $\Gamma \vdash_{\mathtt{Q}} D \setminus (\nu n)\, E : \tau' \blacktriangleright \mathtt{PERMS}$, we have $\Gamma \vdash D$ and $\Gamma \vdash_{\mathtt{Q}} (\nu n)\, E : \tau' \blacktriangleright \mathtt{PERMS}$ by (T-Eval). The latter judgement can be derived only by (T-Restr), hence we have $\Gamma, n : \tau \vdash_{\mathtt{Q}} E : \tau' \blacktriangleright \mathtt{PERMS}$ with $n \notin \mathit{dom}(\Gamma)$. Now we apply Lemma 5.1.3 (Weakening) to derive $\Gamma, n : \tau \vdash D$ from $\Gamma \vdash D$, hence we have $\Gamma, n : \tau \vdash_{\mathtt{Q}} D \setminus E : \tau' \blacktriangleright \mathtt{PERMS}$ by (T-Eval) and we conclude $\Gamma \vdash_{\mathtt{Q}} (\nu n)\, (D \setminus E) : \tau' \blacktriangleright \mathtt{PERMS}$ by (T-Restr);

Case (H-Flip-1): let $[\mathtt{PERMS}]\, (\nu n)\, E \Rrightarrow (\nu n)\, [\mathtt{PERMS}]\, E$. Since $\Gamma \vdash_{\mathtt{Q}} [\mathtt{PERMS}]\, (\nu n)\, E : \tau' \blacktriangleright \mathtt{PERMS}'$, we have $\Gamma \vdash_{\mathtt{PERMS}} (\nu n : \tau)\, E : \tau' \blacktriangleright \mathtt{PERMS}'$ and $\mathtt{PERMS} \sqsubseteq \mathtt{Q}$ by (T-Perms). The latter judgement can be derived only by (T-Restr), hence we have $\Gamma, n : \tau \vdash_{\mathtt{PERMS}} E : \tau' \blacktriangleright \mathtt{PERMS}'$ with $n \notin \mathit{dom}(\Gamma)$. We then get $\Gamma, n : \tau \vdash_{\mathtt{Q}} [\mathtt{PERMS}]\, E : \tau' \blacktriangleright \mathtt{PERMS}'$ by (T-Perms) and we conclude $\Gamma \vdash_{\mathtt{Q}} (\nu n)\, [\mathtt{PERMS}]\, E : \tau' \blacktriangleright \mathtt{PERMS}'$ by (T-Restr);

Case (H-Flip-2): let $[\mathtt{PERMS}]\, (D \setminus E) \Rrightarrow D \setminus [\mathtt{PERMS}]\, E$. Since $\Gamma \vdash_{\mathtt{Q}} [\mathtt{PERMS}]\, (D \setminus E) : \tau \blacktriangleright \mathtt{PERMS}'$, we have $\Gamma \vdash_{\mathtt{PERMS}} D \setminus E : \tau \blacktriangleright \mathtt{PERMS}'$ and $\mathtt{PERMS} \sqsubseteq \mathtt{Q}$ by (T-Perms). The latter judgement can be derived only by (T-Eval), hence we have $\Gamma \vdash D$ and $\Gamma \vdash_{\mathtt{PERMS}} E : \tau \blacktriangleright \mathtt{PERMS}'$. We then get $\Gamma \vdash_{\mathtt{Q}} [\mathtt{PERMS}]\, E : \tau \blacktriangleright \mathtt{PERMS}'$ by (T-Perms) and we conclude $\Gamma \vdash_{\mathtt{Q}} D \setminus [\mathtt{PERMS}]\, E : \tau \blacktriangleright \mathtt{PERMS}'$ by (T-Eval);

Case (H-Comm): let $(D_1 \wedge D_2) \setminus E \Rrightarrow (D_2 \wedge D_1) \setminus E$. Since $\Gamma \vdash_{\mathtt{Q}} (D_1 \wedge D_2) \setminus E : \tau \blacktriangleright \mathtt{PERMS}$, we have $\Gamma \vdash D_1 \wedge D_2$ and $\Gamma \vdash_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{PERMS}$ by (T-Eval). The former judgement can be derived only by (T-Conj), hence we have $\Gamma \vdash D_1$ and $\Gamma \vdash D_2$. We then get $\Gamma \vdash D_2 \wedge D_1$ by (T-Conj) and we conclude $\Gamma \vdash_{\mathtt{Q}} (D_2 \wedge D_1) \setminus E : \tau \blacktriangleright \mathtt{PERMS}$ by (T-Eval). The other direction is analogous.

Case (H-Assoc): let $(D_1 \wedge D_2) \wedge D_3 \setminus E \Rrightarrow D_1 \wedge (D_2 \wedge D_3) \setminus E$. Since $\Gamma \vdash_{\mathtt{Q}} (D_1 \wedge D_2) \wedge D_3 \setminus E : \tau \blacktriangleright \mathtt{PERMS}$, we have $\Gamma \vdash (D_1 \wedge D_2) \wedge D_3$ and $\Gamma \vdash_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{PERMS}$ by (T-Eval). The former judgement can be derived only by (T-Conj), hence we have $\Gamma \vdash D_1 \wedge D_2$ and $\Gamma \vdash D_3$. Again the former judgement can be derived only by (T-Conj), hence we have $\Gamma \vdash D_1$ and $\Gamma \vdash D_2$. We then get $\Gamma \vdash D_2 \wedge D_3$ by (T-Conj) and $\Gamma \vdash D_1 \wedge (D_2 \wedge D_3)$ again by (T-Conj), so we conclude $\Gamma \vdash_{\mathtt{Q}} D_1 \wedge (D_2 \wedge D_3) \setminus E : \tau \blacktriangleright \mathtt{PERMS}$ by (T-Eval). The other direction is analogous.

Case (H-Move): assume $D \setminus (\mathsf{let}\ x = E\ \mathsf{in}\ E') \Rrightarrow \mathsf{let}\ x = (D \setminus E)\ \mathsf{in}\ E'$. Since $\Gamma \vdash_{\mathtt{R}} D \setminus (\mathsf{let}\ x = E\ \mathsf{in}\ E') : \tau \blacktriangleright \mathtt{PERMS}$, we have $\Gamma \vdash D$ and $\Gamma \vdash_{\mathtt{R}} \mathsf{let}\ x = E\ \mathsf{in}\ E' : \tau \blacktriangleright \mathtt{PERMS}$ by (T-Eval). The latter judgement can be derived only by (T-Let), hence we have $\Gamma \vdash_{\mathtt{R}} E : \tau' \blacktriangleright \mathtt{P}$ and $\Gamma, x : \tau' \vdash_{\mathtt{R}} E' : \tau \blacktriangleright \mathtt{Q}$ with $\mathtt{P} \sqcup \mathtt{Q} = \mathtt{PERMS}$ and $x \notin \mathit{dom}(\Gamma)$. We then get $\Gamma \vdash_{\mathtt{R}} D \setminus E : \tau' \blacktriangleright \mathtt{P}$ by (T-Eval) and we conclude $\Gamma \vdash_{\mathtt{R}} \mathsf{let}\ x = (D \setminus E)\ \mathsf{in}\ E' : \tau \blacktriangleright \mathtt{PERMS}$ by (T-Let).

Assume now $\mathsf{let}\ x = (D \setminus E)\ \mathsf{in}\ E' \Rrightarrow D \setminus (\mathsf{let}\ x = E\ \mathsf{in}\ E')$. Since $\Gamma \vdash_{\mathtt{R}} \mathsf{let}\ x = (D \setminus E)\ \mathsf{in}\ E' : \tau \blacktriangleright \mathtt{PERMS}$, we have $\Gamma \vdash_{\mathtt{R}} D \setminus E : \tau' \blacktriangleright \mathtt{P}$ and $\Gamma, x : \tau' \vdash_{\mathtt{R}} E' : \tau \blacktriangleright \mathtt{Q}$ with $\mathtt{P} \sqcup \mathtt{Q} = \mathtt{PERMS}$ and $x \notin \mathit{dom}(\Gamma)$ by (T-Let).The former judgement can be derived only by (T-EVAL), hence we have $\Gamma \vdash D$ and $\Gamma \vdash_{\mathtt{R}} E : \tau' \blacktriangleright \mathtt{P}$. We then get $\Gamma \vdash_{\mathtt{R}} \mathsf{let}\ x = E\ \mathsf{in}\ E' : \tau \blacktriangleright \mathtt{PERMS}$ by (T-LET) and we conclude $\Gamma \vdash_{\mathtt{R}} D \setminus (\mathsf{let}\ x = E\ \mathsf{in}\ E') : \tau \blacktriangleright \mathtt{PERMS}$ by (T-EVAL).

Case (H-CONJ): let $D_1 \setminus (D_2 \setminus E) \Rrightarrow (D_1 \wedge D_2) \setminus E$. Since $\Gamma \vdash_{\mathtt{Q}} D_1 \setminus (D_2 \setminus E) : \tau \blacktriangleright \mathtt{PERMS}$, we have $\Gamma \vdash D_1$ and $\Gamma \vdash_{\mathtt{Q}} D_2 \setminus E : \tau \blacktriangleright \mathtt{PERMS}$ by (T-EVAL). The latter judgement can be derived only by (T-EVAL), hence we have $\Gamma \vdash D_2$ and $\Gamma \vdash_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{PERMS}$. We then get $\Gamma \vdash D_1 \wedge D_2$ by (T-CONJ) and we conclude $\Gamma \vdash_{\mathtt{Q}} (D_1 \wedge D_2) \setminus E : \tau \blacktriangleright \mathtt{PERMS}$ by (T-EVAL).

Assume now $(D_1 \wedge D_2) \setminus E \Rrightarrow D_1 \setminus (D_2 \setminus E)$. Since $\Gamma \vdash_{\mathtt{Q}} (D_1 \wedge D_2) \setminus E : \tau \blacktriangleright \mathtt{PERMS}$, we have $\Gamma \vdash D_1 \wedge D_2$ and $\Gamma \vdash_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{PERMS}$ by (T-EVAL). The former judgement can be derived only by (T-CONJ), hence we have $\Gamma \vdash D_1$ and $\Gamma \vdash D_2$. We then get $\Gamma \vdash_{\mathtt{Q}} D_2 \setminus E : \tau \blacktriangleright \mathtt{PERMS}$ by (T-EVAL) and we conclude $\Gamma \vdash_{\mathtt{Q}} D_1 \setminus (D_2 \setminus E) : \tau \blacktriangleright \mathtt{PERMS}$ again by (T-EVAL).

Case (H-DISTR): assume $[\mathtt{PERMS}]\,\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \Rrightarrow \mathsf{let}\ x = [\mathtt{PERMS}]\,E_1\ \mathsf{in}\ [\mathtt{PERMS}]\,E_2$. Since $\Gamma \vdash_{\mathtt{R}} [\mathtt{PERMS}]\,\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \tau_2 \blacktriangleright \mathtt{PERMS}'$, we have $\Gamma \vdash_{\mathtt{PERMS}} \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \tau_2 \blacktriangleright \mathtt{PERMS}'$ and $\mathtt{PERMS} \sqsubseteq \mathtt{R}$ by (T-PERMS). The latter judgement can be derived only by (T-LET), hence we have $\Gamma \vdash_{\mathtt{PERMS}} E_1 : \tau_1 \blacktriangleright \mathtt{P}$ and $\Gamma, x : \tau_1 \vdash_{\mathtt{PERMS}} E_2 : \tau_2 \blacktriangleright \mathtt{Q}$ with $\mathtt{P} \sqcup \mathtt{Q} = \mathtt{PERMS}'$. We then have $\Gamma \vdash_{\mathtt{R}} [\mathtt{PERMS}]\,E_1 : \tau_1 \blacktriangleright \mathtt{P}$ and $\Gamma, x : \tau_1 \vdash_{\mathtt{R}} [\mathtt{PERMS}]\,E_2 : \tau_2 \blacktriangleright \mathtt{Q}$ by (T-PERMS), hence we conclude $\Gamma \vdash_{\mathtt{R}} \mathsf{let}\ x = [\mathtt{PERMS}]\,E_1\ \mathsf{in}\ [\mathtt{PERMS}]\,E_2 : \tau_2 \blacktriangleright \mathtt{PERMS}'$ by (T-LET).

□

5.2 Proof of subject reduction

Definition 5.2.1 (Permission Lowering). Let $\Gamma \vdash^{\xi}_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{P}$. We define the *permission lowering* of the expression E with respect to the type derivation ξ, written $\xi \cdot E$, by induction on the structure of E:

- $E = [\mathtt{PERMS}]\,E' \Rightarrow \xi \cdot E \triangleq [\mathtt{P} \sqcap \mathtt{PERMS}]\,E'$;
- $E = (\nu n)\,E' \Rightarrow \xi \cdot E \triangleq (\nu n)\,(\xi' \cdot E')$;
- $E = D \setminus E' \Rightarrow \xi \cdot E \triangleq D \setminus (\xi' \cdot E')$;
- $E = (\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2) \Rightarrow \xi \cdot E \triangleq \mathsf{let}\ x = (\xi_1 \cdot E_1)\ \mathsf{in}\ (\xi_2 \cdot E_2)$,

where ξ', ξ_1 and ξ_2 denote the sub-derivations of ξ assigning types to the sub-expressions E', E_1 and E_2, respectively. In all the other cases, we let $\xi \cdot E \triangleq E$.

Lemma 5.2.1 (Deterministic Lowering). *If* $\Gamma \vdash^{\xi}_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{P}$ *and* $\Gamma \vdash^{\xi'}_{\mathtt{Q}'} E : \tau' \blacktriangleright \mathtt{P}'$, *then* $\xi \cdot E = \xi' \cdot E$.

Proof. We first prove the following statement:

$$\text{If } \Gamma \vdash^{\xi}_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{P} \text{ and } \Gamma \vdash^{\xi'}_{\mathtt{Q}} E : \tau' \blacktriangleright \mathtt{P}', \text{ then } \mathtt{P} = \mathtt{P}'.$$

The proof is by induction on the structure of E. The only interesting case is when E is an invocation, i.e., when $E = \overline{u}\langle v \triangleright \mathtt{RECV}\rangle$. Assume then that $\Gamma \vdash^{\xi}_{\mathtt{Q}} \overline{u}\langle v \triangleright \mathtt{RECV}\rangle : \tau \blacktriangleright \mathtt{P}$ and $\Gamma \vdash^{\xi'}_{\mathtt{Q}} \overline{u}\langle v \triangleright \mathtt{RECV}\rangle : \tau' \blacktriangleright \mathtt{P}'$, we perform a case analysis on the last typing rule applied in ξ and ξ':

Case (T-Call)/(T-Call): let $\Gamma \vdash u : \mathsf{Fun}(\mathtt{CALL}, \tau_1 \to \tau_2)^{\mathtt{SECR}}$ among the premises of ξ and $\Gamma \vdash u : \mathsf{Fun}(\mathtt{CALL}', \tau_1' \to \tau_2')^{\mathtt{SECR}'}$ among the premises of ξ', then we have $\mathtt{P} = \mathtt{CALL} \sqcup \mathtt{SECR}$ and $\mathtt{P}' = \mathtt{CALL}' \sqcup \mathtt{SECR}'$. The conclusion follows by Proposition 5.1.1 (Uniqueness of Function Types);

Case (T-Call-Un)/(T-Call-Un): the case is immediate, since $\mathtt{P} = \mathtt{P}' = \bot$;

Case (T-Fail)/(T-Fail): the case is immediate, since $\mathtt{P} = \mathtt{P}' = \mathtt{Q}$;

Case (T-Call)/(T-Fail): let $\Gamma \vdash u : \mathsf{Fun}(\mathtt{CALL}, \tau_1 \to \tau_2)^{\mathtt{SECR}}$ among the premises of ξ and let $\Gamma \vdash u : \mathsf{Fun}(\mathtt{CALL}', \tau_1' \to \tau_2')^{\mathtt{SECR}'}$ among the premises of ξ'. Since we have $\mathtt{CALL} \sqcup \mathtt{SECR} \sqsubseteq \mathtt{Q}$ in ξ, we know that $\mathtt{CALL} \sqsubseteq \mathtt{Q}$ by transitivity; however, we also have $\mathtt{CALL}' \not\sqsubseteq \mathtt{Q}$ in ξ', so we get a contradiction by Proposition 5.1.1 (Uniqueness of Function Types);

Case (T-Call)/(T-Call-Un): let $\Gamma \vdash u : \mathsf{Fun}(\mathtt{CALL}, \tau_1 \to \tau_2)^{\mathtt{SECR}}$ among the premises of ξ, we have $\mathtt{P} = \mathtt{CALL} \sqcup \mathtt{SECR} \sqsubseteq \mathtt{Q}$. But note that $\mathtt{Q} = \bot$, otherwise we could not apply rule (T-Call-Un), hence $\mathtt{P} = \bot$ by anti-symmetry. Since $\mathtt{P}' = \bot$, we conclude;

Case (T-Fail)/(T-Call-Un): in this case we have $\mathtt{P} = \mathtt{Q}$. But note that $\mathtt{Q} = \bot$, otherwise we could not apply rule (T-Call-Un). Since $\mathtt{P}' = \bot$, we conclude.

The symmetric cases are analogous.

The main statement is proved again by induction on the structure of E. The only interesting case is when E is a permission assignment, i.e., when $E = [\mathtt{PERMS}]\, E'$. Assume then $\Gamma \vdash^{\xi}_{\mathtt{Q}} [\mathtt{PERMS}]\, E' : \tau \blacktriangleright \mathtt{P}$ and $\Gamma \vdash^{\xi'}_{\mathtt{Q}'} [\mathtt{PERMS}]\, E' : \tau' \blacktriangleright \mathtt{P}'$, in this case both ξ and ξ' are concluded by an application of rule (T-Perms), hence we know that $\Gamma \vdash_{\mathtt{PERMS}} E' : \tau \blacktriangleright \mathtt{P}$ among the premises of ξ and $\Gamma \vdash_{\mathtt{PERMS}} E' : \tau' \blacktriangleright \mathtt{P}'$ among the premises of ξ'. By the previous result we have $\mathtt{P} = \mathtt{P}'$, thus $\xi \cdot E = [\mathtt{P} \sqcap \mathtt{PERMS}]\, E' = [\mathtt{P}' \sqcap \mathtt{PERMS}]\, E' = \xi' \cdot E$. □

Notation 5.2.2. By Lemma 5.2.1 (Deterministic Lowering), for any well-typed expression E we can write $\Gamma \cdot E$ to stand for $\xi \cdot E$ for an arbitrarily chosen type derivation ξ such that $\Gamma \vdash^{\xi}_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{P}$ for some $\mathtt{P}$, $\mathtt{Q}$ and τ.

Definition 5.2.3 (Expression Ordering)**.** We overload the symbol $\sqsubseteq$ to denote the smallest pre-order on expressions closed under the following inference rules:

$$\frac{\texttt{PERMS}_1 \sqsubseteq \texttt{PERMS}_2}{[\texttt{PERMS}_1]\, E \sqsubseteq [\texttt{PERMS}_2]\, E} \qquad \frac{E \sqsubseteq E'}{(\nu n)\, E \sqsubseteq (\nu n)\, E'} \qquad \frac{E \sqsubseteq E'}{D \setminus E \sqsubseteq D \setminus E'}$$

$$\frac{E_1 \sqsubseteq E_1' \qquad E_2 \sqsubseteq E_2'}{\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \sqsubseteq \mathsf{let}\ x = E_1'\ \mathsf{in}\ E_2'}$$

Proposition 5.2.2 (Soundness of Lowering)**.** *For any E such that $\Gamma \vdash E : \tau \blacktriangleright \texttt{P}$, we have $E \sqsupseteq \Gamma \cdot E$.*

Proof. By induction on the structure of E. □

Lemma 5.2.3 (Lowering Respects Heating)**.** *Let $\Gamma \vdash E : \tau \blacktriangleright \texttt{P}$. If $E \Rrightarrow E'$, then $\Gamma \cdot E'$ is defined and $\Gamma \cdot E \Rrightarrow E''$ for some $E'' \sqsupseteq \Gamma \cdot E'$.*

Proof. First of all, we note that $\Gamma \vdash E' : \tau \blacktriangleright \texttt{P}$ by Lemma 5.1.5 (Heating Preserves Typing), hence $\Gamma \cdot E'$ is defined. We then proceed by induction on the derivation of $E \Rrightarrow E'$:

Case (H-Eval): let $D \setminus E \Rrightarrow D \setminus E'$ by the premise $E \Rrightarrow E'$. Since $\Gamma \vdash D \setminus E : \tau \blacktriangleright \texttt{P}$, we have $\Gamma \vdash D$ and $\Gamma \vdash E : \tau \blacktriangleright \texttt{P}$ by (T-Eval). By inductive hypothesis $\Gamma \cdot E \Rrightarrow E''$ for some $E'' \sqsupseteq \Gamma \cdot E'$, hence we have:

$$\begin{aligned} \Gamma \cdot (D \setminus E) &\triangleq D \setminus (\Gamma \cdot E) \\ &\Rrightarrow D \setminus E'' \\ &\sqsupseteq D \setminus (\Gamma \cdot E') \\ &\triangleq \Gamma \cdot (D \setminus E'). \end{aligned}$$

Case (H-Let): assume $\mathsf{let}\ x = E\ \mathsf{in}\ E'' \Rrightarrow \mathsf{let}\ x = E'\ \mathsf{in}\ E''$ by the premise $E \Rrightarrow E'$. Since $\Gamma \vdash \mathsf{let}\ x = E\ \mathsf{in}\ E'' : \tau \blacktriangleright \texttt{PERMS}$, we have $\Gamma \vdash E : \tau' \blacktriangleright \texttt{P}$ and $\Gamma, x : \tau' \vdash E'' : \tau \blacktriangleright \texttt{Q}$ with $\texttt{P} \sqcup \texttt{Q} = \texttt{PERMS}$ by (T-Let). By inductive hypothesis $\Gamma \cdot E \Rrightarrow \hat{E}$ for some $\hat{E} \sqsupseteq \Gamma \cdot E'$, hence we have:

$$\begin{aligned} \Gamma \cdot (\mathsf{let}\ x = E\ \mathsf{in}\ E'') &\triangleq \mathsf{let}\ x = (\Gamma \cdot E)\ \mathsf{in}\ (\Gamma, x : \tau) \cdot E'' \\ &\Rrightarrow \mathsf{let}\ x = \hat{E}\ \mathsf{in}\ (\Gamma, x : \tau) \cdot E'' \\ &\sqsupseteq \mathsf{let}\ x = (\Gamma \cdot E')\ \mathsf{in}\ (\Gamma, x : \tau) \cdot E'' \\ &\triangleq \Gamma \cdot (\mathsf{let}\ x = E'\ \mathsf{in}\ E''). \end{aligned}$$

Case (H-Restr): let $(\nu n)\, E \Rrightarrow (\nu n)\, E'$ by the premise $E \Rrightarrow E'$. Since $\Gamma \vdash (\nu n)\, E : \tau' \blacktriangleright \texttt{PERMS}$, we have $\Gamma, n : \tau \vdash E : \tau' \blacktriangleright \texttt{PERMS}$ by (T-Restr). By inductive hypothesis $(\Gamma, n:\tau)\cdot E \Rrightarrow E''$ for some $E'' \sqsupseteq (\Gamma, n:\tau)\cdot E'$, hence we have:

$$\begin{aligned}\Gamma\cdot((\nu n)\,E) &\triangleq (\nu n)\,((\Gamma, n:\tau)\cdot E)\\ &\Rrightarrow (\nu n)\,E''\\ &\sqsupseteq (\nu n)\,((\Gamma, n:\tau)\cdot E')\\ &\triangleq \Gamma\cdot((\nu n)\,E').\end{aligned}$$

Case (H-Extr-1): assume $\mathsf{let}\ x = (\nu n)\,E_1\ \mathsf{in}\ E_2 \Rrightarrow (\nu n)\,(\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2)$ with $n \notin \mathit{fn}(E_2)$. Since $\Gamma \vdash \mathsf{let}\ x = (\nu n)\,E_1\ \mathsf{in}\ E_2 : \tau_2 \blacktriangleright \mathtt{R}$, we have $\Gamma \vdash (\nu n)\,E_1 : \tau_1 \blacktriangleright \mathtt{P}$ and $\Gamma, x:\tau_1 \vdash E_2 : \tau_2 \blacktriangleright \mathtt{Q}$ with $\mathtt{P}\sqcup\mathtt{Q} = \mathtt{R}$ and $x \notin \mathit{dom}(\Gamma)$ by (T-Let). The former judgement can be derived only by (T-Restr), hence we have $\Gamma, n:\tau \vdash E_1 : \tau_1 \blacktriangleright \mathtt{P}$ with $n \notin \mathit{dom}(\Gamma)$. Now we apply Lemma 5.1.3 (Weakening) to derive $\Gamma, n:\tau, x:\tau_1 \vdash E_2 : \tau_2 \blacktriangleright \mathtt{Q}$ from $\Gamma, x:\tau_1 \vdash E_2 : \tau_2 \blacktriangleright \mathtt{Q}$. Notice that the lemma also implies that $(\Gamma, x:\tau_1)\cdot E_2 = (\Gamma, n:\tau, x:\tau_1)\cdot E_2$, hence we have:

$$\begin{aligned}\Gamma\cdot(\mathsf{let}\ x = (\nu n)\,E_1\ \mathsf{in}\ E_2) &\triangleq \mathsf{let}\ x = (\Gamma\cdot(\nu n)\,E_1)\ \mathsf{in}\ (\Gamma, x:\tau_1)\cdot E_2\\ &\triangleq \mathsf{let}\ x = (\nu n)\,((\Gamma, n:\tau)\cdot E_1)\ \mathsf{in}\ (\Gamma, x:\tau_1)\cdot E_2\\ &\Rrightarrow (\nu n)\,(\mathsf{let}\ x = ((\Gamma, n:\tau)\cdot E_1)\ \mathsf{in}\ (\Gamma, x:\tau_1)\cdot E_2)\\ &= (\nu n)\,(\mathsf{let}\ x = ((\Gamma, n:\tau)\cdot E_1)\ \mathsf{in}\ (\Gamma, n:\tau, x:\tau_1)\cdot E_2)\\ &\triangleq \Gamma\cdot((\nu n)\,(\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2)).\end{aligned}$$

Case (H-Extr-2): let $D\setminus(\nu n)\,E \Rrightarrow (\nu n)\,(D\setminus E)$ with $n \notin \mathit{fn}(D)$. Given that $\Gamma \vdash D\setminus(\nu n)\,E : \tau' \blacktriangleright \mathtt{PERMS}$, we have $\Gamma \vdash D$ and $\Gamma \vdash (\nu n)\,E : \tau' \blacktriangleright \mathtt{PERMS}$ by (T-Eval). The latter judgement can be derived only by (T-Restr), hence we have $\Gamma, n:\tau \vdash E : \tau' \blacktriangleright \mathtt{PERMS}$ with $n \notin \mathit{dom}(\Gamma)$. Hence, we have:

$$\begin{aligned}\Gamma\cdot(D\setminus(\nu n)\,E) &\triangleq D\setminus(\Gamma\cdot(\nu n)\,E)\\ &\triangleq D\setminus(\nu n)\,((\Gamma, n:\tau)\cdot E)\\ &\Rrightarrow (\nu n)\,(D\setminus((\Gamma, n:\tau)\cdot E))\\ &\triangleq \Gamma\cdot((\nu n)\,(D\setminus E))\end{aligned}$$

Case (H-Flip-1): let $[\mathtt{PERMS}]\,(\nu n)\,E \Rrightarrow (\nu n)\,[\mathtt{PERMS}]\,E$. Since $\Gamma \vdash_{\mathtt{R}} [\mathtt{PERMS}]\,(\nu n)\,E : \tau' \blacktriangleright \mathtt{Q}$, we have $\Gamma \vdash_{\mathtt{PERMS}} (\nu n:\tau)\,E : \tau' \blacktriangleright \mathtt{Q}$ and $\mathtt{PERMS} \sqsubseteq \mathtt{R}$ by (T-Perms). The latter judgement can be derived only by (T-Restr), hence we have $\Gamma, n:\tau \vdash_{\mathtt{PERMS}} E : \tau' \blacktriangleright \mathtt{Q}$ with $n \notin \mathit{dom}(\Gamma)$. We then get $\Gamma, n:\tau \vdash_{\mathtt{R}} [\mathtt{PERMS}]\,E : \tau' \blacktriangleright \mathtt{Q}$ by (T-Perms). Hence, we have:

$$\begin{aligned}\Gamma\cdot([\mathtt{PERMS}]\,(\nu n)\,E) &\triangleq [\mathtt{PERMS}\sqcap\mathtt{Q}]\,(\nu n)\,E\\ &\Rrightarrow (\nu n)\,[\mathtt{PERMS}\sqcap\mathtt{Q}]\,E\\ &\triangleq (\nu n)\,((\Gamma, n:\tau)\cdot[\mathtt{PERMS}]\,E)\\ &\triangleq \Gamma\cdot((\nu n)\,[\mathtt{PERMS}]\,E).\end{aligned}$$

Case (H-Flip-2): let $[\texttt{PERMS}]\,(D \setminus E) \Rrightarrow D \setminus [\texttt{PERMS}]\,E$. Since $\Gamma \vdash_{\texttt{R}} [\texttt{PERMS}]\,(D \setminus E) : \tau \blacktriangleright \texttt{Q}$, we have $\Gamma \vdash_{\texttt{PERMS}} D \setminus E : \tau \blacktriangleright \texttt{Q}$ and $\texttt{PERMS} \sqsubseteq \texttt{R}$ by (T-Perms). The latter judgement can be derived only by (T-Eval), hence we have $\Gamma \vdash D$ and $\Gamma \vdash_{\texttt{PERMS}} E : \tau \blacktriangleright \texttt{Q}$. We then get $\Gamma \vdash_{\texttt{R}} [\texttt{PERMS}]\,E : \tau \blacktriangleright \texttt{Q}$ by (T-Perms). Hence, we have:

$$\begin{aligned}
\Gamma \cdot ([\texttt{PERMS}]\,(D \setminus E)) &\triangleq [\texttt{Q} \sqcap \texttt{PERMS}]\,(D \setminus E) \\
&\Rrightarrow D \setminus [\texttt{Q} \sqcap \texttt{PERMS}]\,E \\
&\triangleq D \setminus (\Gamma \cdot ([\texttt{PERMS}]\,E)) \\
&\triangleq \Gamma \cdot (D \setminus [\texttt{PERMS}]\,E).
\end{aligned}$$

Case (H-Comm): let $(D_1 \wedge D_2) \setminus E \Rrightarrow (D_2 \wedge D_1) \setminus E$. Since $\Gamma \vdash (D_1 \wedge D_2) \setminus E : \tau \blacktriangleright \texttt{PERMS}$, we have $\Gamma \vdash D_1 \wedge D_2$ and $\Gamma \vdash E : \tau \blacktriangleright \texttt{PERMS}$ by (T-Eval). Hence, we have:

$$\begin{aligned}
\Gamma \cdot ((D_1 \wedge D_2) \setminus E) &\triangleq (D_1 \wedge D_2) \setminus (\Gamma \cdot E) \\
&\Rrightarrow (D_2 \wedge D_1) \setminus (\Gamma \cdot E) \\
&\triangleq \Gamma \cdot ((D_2 \wedge D_1) \setminus E).
\end{aligned}$$

The other direction is analogous.

Case (H-Assoc): let $(D_1 \wedge D_2) \wedge D_3 \setminus E \Rrightarrow D_1 \wedge (D_2 \wedge D_3) \setminus E$. Since $\Gamma \vdash (D_1 \wedge D_2) \wedge D_3 \setminus E : \tau \blacktriangleright \texttt{PERMS}$, we have $\Gamma \vdash (D_1 \wedge D_2) \wedge D_3$ and $\Gamma \vdash E : \tau \blacktriangleright \texttt{PERMS}$ by (T-Eval). Hence, we have:

$$\begin{aligned}
\Gamma \cdot ((D_1 \wedge D_2) \wedge D_3 \setminus E) &\triangleq (D_1 \wedge D_2) \wedge D_3 \setminus (\Gamma \cdot E) \\
&\Rrightarrow D_1 \wedge (D_2 \wedge D_3) \setminus (\Gamma \cdot E) \\
&\triangleq \Gamma \cdot (D_1 \wedge (D_2 \wedge D_3) \setminus E).
\end{aligned}$$

The other direction is analogous.

Case (H-Conj): let $D_1 \setminus (D_2 \setminus E) \Rrightarrow (D_1 \wedge D_2) \setminus E$. Since $\Gamma \vdash D_1 \setminus (D_2 \setminus E) : \tau \blacktriangleright \texttt{PERMS}$, we have $\Gamma \vdash D_1$ and $\Gamma \vdash D_2 \setminus E : \tau \blacktriangleright \texttt{PERMS}$ by (T-Eval). The latter judgement can be derived only by (T-Eval), hence we have $\Gamma \vdash D_2$ and $\Gamma \vdash E : \tau \blacktriangleright \texttt{PERMS}$. Hence, we have:

$$\begin{aligned}
\Gamma \cdot (D_1 \setminus (D_2 \setminus E)) &\triangleq D_1 \setminus D_2 \setminus (\Gamma \cdot E) \\
&\Rrightarrow D_1 \wedge D_2 \setminus (\Gamma \cdot E) \\
&\triangleq \Gamma \cdot ((D_1 \wedge D_2) \setminus E).
\end{aligned}$$

The other direction is similar.

Case (H-Move): assume $D \setminus (\mathsf{let}\ x = E\ \mathsf{in}\ E') \Rrightarrow \mathsf{let}\ x = (D \setminus E)\ \mathsf{in}\ E'$. Since $\Gamma \vdash D \setminus (\mathsf{let}\ x = E\ \mathsf{in}\ E') : \tau \blacktriangleright \texttt{R}$, we have $\Gamma \vdash D$ and $\Gamma \vdash \mathsf{let}\ x = E\ \mathsf{in}\ E' : \tau \blacktriangleright \texttt{R}$ by

(T-EVAL). The latter judgement can be derived only by (T-LET), hence we have $\Gamma \vdash E : \tau' \blacktriangleright \mathtt{P}$ and $\Gamma, x : \tau' \vdash E' : \tau \blacktriangleright \mathtt{Q}$ with $\mathtt{P} \sqcup \mathtt{Q} = \mathtt{R}$ and $x \notin \mathit{dom}(\Gamma)$. Hence, we have:

$$\begin{aligned}\Gamma \cdot (D \setminus (\mathsf{let}\ x = E\ \mathsf{in}\ E')) &\triangleq D \setminus (\Gamma \cdot (\mathsf{let}\ x = E\ \mathsf{in}\ E'))\\ &\triangleq D \setminus \mathsf{let}\ x = (\Gamma \cdot E)\ \mathsf{in}\ (\Gamma, x : \tau') \cdot E'\\ &\Rrightarrow \mathsf{let}\ x = D \setminus (\Gamma \cdot E)\ \mathsf{in}\ (\Gamma, x : \tau') \cdot E'\\ &\triangleq \mathsf{let}\ x = \Gamma \cdot (D \setminus E)\ \mathsf{in}\ (\Gamma, x : \tau') \cdot E'\\ &\triangleq \Gamma \cdot (\mathsf{let}\ x = (D \setminus E)\ \mathsf{in}\ E').\end{aligned}$$

The other direction is similar.

Case (H-DISTR): assume $[\mathtt{PERMS}]\,\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \Rrightarrow \mathsf{let}\ x = [\mathtt{PERMS}]\,E_1\ \mathsf{in}\ [\mathtt{PERMS}]\,E_2$ with $\Gamma \vdash [\mathtt{PERMS}]\,\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \tau_2 \blacktriangleright \mathtt{PERMS}'$. The judgement must have been derived by (T-PERMS), hence we know $\Gamma \vdash_{\mathtt{PERMS}} \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \tau_2 \blacktriangleright \mathtt{PERMS}'$. This can be derived only by (T-LET), so we have $\Gamma \vdash_{\mathtt{PERMS}} E_1 : \tau_1 \blacktriangleright \mathtt{P}$ and $\Gamma, x : \tau_1 \vdash_{\mathtt{PERMS}} E_2 : \tau_2 \blacktriangleright \mathtt{Q}$ with $\mathtt{P} \sqcup \mathtt{Q} = \mathtt{PERMS}'$. We can then apply (T-PERMS) to derive $\Gamma \vdash_{\mathtt{PERMS}} [\mathtt{PERMS}]\,E_1 : \tau_1 \blacktriangleright \mathtt{P}$ and $\Gamma, x : \tau_1 \vdash_{\mathtt{PERMS}} [\mathtt{PERMS}]\,E_2 : \tau_2 \blacktriangleright \mathtt{Q}$.

Now we notice that we have:

$$\begin{aligned}\Gamma \cdot ([\mathtt{PERMS}]\,\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2) &\triangleq [(\mathtt{P} \sqcup \mathtt{Q}) \sqcap \mathtt{PERMS}]\,\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2\\ &\Rrightarrow \mathsf{let}\ x = [(\mathtt{P} \sqcup \mathtt{Q}) \sqcap \mathtt{PERMS}]\,E_1\ \mathsf{in}\ [(\mathtt{P} \sqcup \mathtt{Q}) \sqcap \mathtt{PERMS}]\,E_2\\ &\sqsupseteq \mathsf{let}\ x = [\mathtt{P} \sqcap \mathtt{PERMS}]\,E_1\ \mathsf{in}\ [\mathtt{Q} \sqcap \mathtt{PERMS}]\,E_2\\ &\triangleq \mathsf{let}\ x = (\Gamma \cdot [\mathtt{PERMS}]\,E_1)\ \mathsf{in}\ ((\Gamma, x : \tau_1) \cdot [\mathtt{PERMS}]\,E_2)\\ &\triangleq \Gamma \cdot (\mathsf{let}\ x = [\mathtt{PERMS}]\,E_1\ \mathsf{in}\ [\mathtt{PERMS}]\,E_2).\end{aligned}$$

□

Lemma 5.2.4 (Monotonicity of Heating). *If $E_1 \Rrightarrow E_2$ and $E_1 \sqsubseteq E_1'$, then $E_1' \Rrightarrow E_2'$ for some $E_2' \sqsupseteq E_2$.*

Proof. By a straightforward induction on the derivation of $E_1 \Rrightarrow E_2$. □

Lemma 5.2.5 (Monotonicity of Reduction). *If $E_1 \xrightarrow{\alpha}_{ipc} E_2$ and $E_1 \sqsubseteq E_1'$, then $E_1' \xrightarrow{\alpha}_{ipc} E_2'$ for some $E_2' \sqsupseteq E_2$.*

Proof. By a straightforward induction on the derivation of $E_1 \xrightarrow{\alpha}_{ipc} E_2$. □

Proposition 5.2.6 (Monotonicity of Typing). *If $\Gamma \vdash_{\mathtt{Q}} [\mathtt{PERMS}]\,E : \tau \blacktriangleright \mathtt{P}$ and $\mathtt{Q} \sqsubseteq \mathtt{R}$, then $\Gamma \vdash_{\mathtt{R}} [\mathtt{PERMS}]\,E : \tau \blacktriangleright \mathtt{P}$.*

Proof. Since $\Gamma \vdash_{\mathtt{Q}} [\mathtt{PERMS}]\,E : \tau \blacktriangleright \mathtt{P}$ can be derived only by (T-PERMS), we know that $\Gamma \vdash_{\mathtt{PERMS}} E : \tau \blacktriangleright \mathtt{P}$ and $\mathtt{PERMS} \sqsubseteq \mathtt{Q}$. Hence, $\mathtt{PERMS} \sqsubseteq \mathtt{R}$ by transitivity and we get $\Gamma \vdash_{\mathtt{R}} [\mathtt{PERMS}]\,E : \tau \blacktriangleright \mathtt{P}$ again by (T-PERMS). □

Theorem 5.2.7 (Simulation-Aware Subject Reduction). *If* $\Gamma \vdash_{\top} E : \tau \blacktriangleright \mathtt{PERMS}$ *and* $E \xrightarrow{\alpha} E'$, *then* $\Gamma \vdash_{\top} E' : \tau \blacktriangleright \mathtt{PERMS}'$ *for some* $\mathtt{PERMS}' \sqsubseteq \mathtt{PERMS}$. *Moreover, there exists* E'' *such that* $\Gamma \cdot E \xrightarrow{\alpha}_{ipc} E''$ *and* $E'' \sqsupseteq \Gamma \cdot E'$.

Proof. By induction on the derivation of $E \xrightarrow{\alpha} E'$:

Case (R-CALL): assume $\mathsf{def}\, n^{\ell} = \lambda(x \lhd \mathtt{CALL}).[\mathtt{PERMS}']\, E \setminus [\mathtt{PERMS}]\, \overline{n}\langle m \rhd \mathtt{RECV}\rangle \xrightarrow{\ell} [\mathtt{PERMS}']\, E\{m/x\}$ with:

(1) $\mathtt{CALL} \sqsubseteq \mathtt{PERMS}$

(2) $\mathtt{RECV} \sqsubseteq \mathtt{PERMS}'$.

By hypothesis we know that:

$$\Gamma \vdash_{\top} \mathsf{def}\, n = \lambda(x \lhd \mathtt{CALL}).[\mathtt{PERMS}']\, E \setminus [\mathtt{PERMS}]\, \overline{n}\langle m \rhd \mathtt{RECV}\rangle : \tau \blacktriangleright \mathtt{P},$$

which must follow by an instance of (T-EVAL). Hence, we know that $\Gamma \vdash \mathsf{def}\, n^{\ell} = \lambda(x \lhd \mathtt{CALL}).[\mathtt{PERMS}']\, E$ and $\Gamma \vdash_{\top} [\mathtt{PERMS}]\, \overline{n}\langle m \rhd \mathtt{RECV}\rangle : \tau \blacktriangleright \mathtt{P}$ must hold. We perform a case analysis on how these latter two judgements are derived.

If $\Gamma \vdash \mathsf{def}\, n^{\ell} = \lambda(x \lhd \mathtt{CALL}).[\mathtt{PERMS}']\, E$ was derived by (T-DEF), we know that:

(3) $\Gamma \vdash n : \mathsf{Fun}(\mathtt{CALL}, \tau_n \to \tau_n')^{\mathtt{SECR}}$

(4) $\Gamma, x : \tau_n \vdash_{\top} [\mathtt{PERMS}']\, E : \tau_n' \blacktriangleright \mathtt{Q}'$ with $x \notin dom(\Gamma)$

(5) $\mathtt{Q}' \sqsubseteq \mathtt{CALL} \sqcup \mathtt{SECR}$

(6) $\mathtt{CALL} \sqcup \mathtt{SECR} = \bot \Rightarrow \Gamma, x : \mathsf{Un} \vdash_{\top} [\mathtt{PERMS}']\, E : \mathsf{Un} \blacktriangleright \bot$ with $x \notin dom(\Gamma)$.

We distinguish three cases, according to the rule used to derive $\Gamma \vdash_{\top} [\mathtt{PERMS}]\, \overline{n}\langle m \rhd \mathtt{RECV}\rangle : \tau \blacktriangleright \mathtt{P}$. If the latter judgement was derived by (T-CALL) after an application of (T-PERMS), then we know that:

(7) $\Gamma \vdash n : \mathsf{Fun}(\mathtt{CALL}', \hat{\tau}_n \to \hat{\tau}_n')^{\mathtt{SECR}'}$

(8) $\Gamma \vdash m : \hat{\tau}_n$

(9) $\mathtt{CALL}' \sqcup \mathtt{SECR}' \sqsubseteq \mathtt{PERMS}$

(10) $\mathtt{P} = \mathtt{CALL}' \sqcup \mathtt{SECR}'$.

By Proposition 5.1.1 (Uniqueness of Function Types), we know that $\hat{\tau}_n = \tau_n$, $\hat{\tau}_n' = \tau_n' = \tau$, $\mathtt{CALL} = \mathtt{CALL}'$ and $\mathtt{SECR} = \mathtt{SECR}'$. By (4) and (8), using Lemma 5.1.4 (Substitution), we then get $\Gamma \vdash_{\top} [\mathtt{PERMS}']\, E\{m/x\} : \tau \blacktriangleright \mathtt{Q}'$. Notice also that $\mathtt{Q}' \sqsubseteq \mathtt{P}$

by (5) and (10). Now we note that:

$$\begin{aligned}
&\Gamma \cdot (\mathsf{def}\, n^{\ell} = \lambda(x \triangleleft \mathtt{CALL}).[\mathtt{PERMS}']\, E \setminus [\mathtt{PERMS}]\, \overline{n}\langle m \triangleright \mathtt{RECV}\rangle) \\
&\triangleq \mathsf{def}\, n^{\ell} = \lambda(x \triangleleft \mathtt{CALL}).[\mathtt{PERMS}']\, E \setminus [(\mathtt{CALL} \sqcup \mathtt{SECR}) \sqcap \mathtt{PERMS}]\, \overline{n}\langle m \triangleright \mathtt{RECV}\rangle \\
&= \mathsf{def}\, n^{\ell} = \lambda(x \triangleleft \mathtt{CALL}).[\mathtt{PERMS}']\, E \setminus [\mathtt{CALL} \sqcup \mathtt{SECR}]\, \overline{n}\langle m \triangleright \mathtt{RECV}\rangle && \text{by (9)} \\
&\xrightarrow{\ell}_{ipc} [(\mathtt{CALL} \sqcup \mathtt{SECR}) \sqcap \mathtt{PERMS}']\, E\{m/x\} \\
&\sqsupseteq [\mathtt{Q}' \sqcap \mathtt{PERMS}']\, E\{m/x\} && \text{by } \mathtt{Q}' \sqsubseteq \mathtt{P} \\
&\triangleq \Gamma \cdot ([\mathtt{PERMS}']\, E\{m/x\}),
\end{aligned}$$

where the reduction step can be performed, since $\mathtt{CALL} \sqsubseteq \mathtt{CALL} \sqcup \mathtt{SECR}$.

Assume then that $\Gamma \vdash_{\top} [\mathtt{PERMS}]\, \overline{n}\langle m \triangleright \mathtt{RECV}\rangle : \tau \blacktriangleright \mathtt{P}$ was derived by (T-Call-Un) after an application of (T-Perms), then we know that:

(11) $\Gamma \vdash n : \mathsf{Un}$

(12) $\Gamma \vdash m : \mathsf{Un}$

(13) $\tau = \mathsf{Un}$

(14) $\mathtt{P} = \bot$

(15) $\mathtt{PERMS} = \bot$.

Since (3) and (11) hold, by Proposition 5.1.2 (Soundness of Secrecy Levels) we know that $\mathtt{SECR} = \bot$. Since hypothesis (1) states $\mathtt{CALL} \sqsubseteq \mathtt{PERMS}$ and (15) holds true, we know that $\mathtt{CALL} = \bot$ by anti-symmetry, so we have $\mathtt{CALL} \sqcup \mathtt{SECR} = \bot$. By (6) we can then get $\Gamma, x : \mathsf{Un} \vdash_{\top} [\mathtt{PERMS}']\, E : \mathsf{Un} \blacktriangleright \bot$, hence, by (12) and Lemma 5.1.4 (Substitution), we get $\Gamma \vdash_{\top} [\mathtt{PERMS}']\, E\{m/x\} : \tau \blacktriangleright \bot$. Now we note that:

$$\begin{aligned}
&\Gamma \cdot (\mathsf{def}\, n^{\ell} = \lambda(x \triangleleft \mathtt{CALL}).[\mathtt{PERMS}']\, E \setminus [\mathtt{PERMS}]\, \overline{n}\langle m \triangleright \mathtt{RECV}\rangle) \\
&= \Gamma \cdot (\mathsf{def}\, n^{\ell} = \lambda(x \triangleleft \mathtt{CALL}).[\mathtt{PERMS}']\, E \setminus [\bot]\, \overline{n}\langle m \triangleright \mathtt{RECV}\rangle) && \text{by (15)} \\
&\triangleq \mathsf{def}\, n^{\ell} = \lambda(x \triangleleft \mathtt{CALL}).[\mathtt{PERMS}']\, E \setminus [\bot]\, \overline{n}\langle m \triangleright \mathtt{RECV}\rangle \\
&\xrightarrow{\ell}_{ipc} [\bot \sqcap \mathtt{PERMS}']\, E\{m/x\} \\
&= [\bot]\, E\{m/x\} \\
&\triangleq \Gamma \cdot ([\mathtt{PERMS}']\, E\{m/x\}),
\end{aligned}$$

where the reduction step can be performed, since we showed that $\mathtt{CALL} = \bot$.

Finally, assume that $\Gamma \vdash_{\top} [\mathtt{PERMS}]\, \overline{n}\langle m \triangleright \mathtt{RECV}\rangle : \tau \blacktriangleright \mathtt{P}$ was derived by (T-Fail) after an application of (T-Perms), then we know that:

(16) $\Gamma \vdash n : \mathsf{Fun}(\mathtt{CALL}', \hat{\tau}_n \to \hat{\tau}'_n)^{\mathtt{SECR}'}$

(17) $\mathtt{CALL}' \not\sqsubseteq \mathtt{PERMS}$.

Since (3) and (16) hold, by Proposition 5.1.1 (Uniqueness of Function Types) we know that $\mathtt{CALL} \not\sqsubseteq \mathtt{PERMS}$, but this is in contradiction with $\mathtt{CALL} \sqsubseteq \mathtt{PERMS}$ from hypothesis (1), hence the case is trivial.

Let us now consider the case when $\Gamma \vdash \mathsf{def}\, n^{\ell} = \lambda(x \lhd \mathtt{CALL}).[\mathtt{PERMS}']\, E$ was derived by (T-Def-Un). In this case we know that:

(18) $\Gamma \vdash n : \mathsf{Un}$

(19) $\Gamma, x : \mathsf{Un} \vdash_{\bot} [\mathtt{PERMS}']\, E : \mathsf{Un} \blacktriangleright \bot$ with $x \notin \mathit{dom}(\Gamma)$.

Note that (19) can be derived only after an application of (T-Perms), which implies $\mathtt{PERMS}' \sqsubseteq \bot$. By anti-symmetry, we then get:

(20) $\mathtt{PERMS}' = \bot$.

We distinguish three cases, according to the rule used to derive $\Gamma \vdash_{\top} [\mathtt{PERMS}]\, \overline{n}\langle m \rhd \mathtt{RECV}\rangle : \tau \blacktriangleright \mathtt{P}$. If the latter judgement was derived by (T-Call-Un) after an application of (T-Perms), then we know that:

(21) $\Gamma \vdash m : \mathsf{Un}$

(22) $\tau = \mathsf{Un}$

(23) $\mathtt{P} = \bot$

(24) $\mathtt{PERMS} = \bot$.

By (19) and (21), using Lemma 5.1.4 (Substitution), we get $\Gamma \vdash_{\bot} [\mathtt{PERMS}']\, E\{m/x\} : \tau \blacktriangleright \bot$, hence we get $\Gamma \vdash_{\top} [\mathtt{PERMS}']\, E\{m/x\} : \tau \blacktriangleright \bot$ by Proposition 5.2.6 (Monotonicity of Typing). Now we note that:

$$\begin{aligned}
&\Gamma \cdot (\mathsf{def}\, n^{\ell} = \lambda(x \lhd \mathtt{CALL}).[\mathtt{PERMS}']\, E \setminus [\mathtt{PERMS}]\, \overline{n}\langle m \rhd \mathtt{RECV}\rangle)\\
&= \Gamma \cdot (\mathsf{def}\, n^{\ell} = \lambda(x \lhd \mathtt{CALL}).[\mathtt{PERMS}']\, E \setminus [\bot]\, \overline{n}\langle m \rhd \mathtt{RECV}\rangle) && \text{by (24)}\\
&\triangleq \mathsf{def}\, n^{\ell} = \lambda(x \lhd \mathtt{CALL}).[\mathtt{PERMS}']\, E \setminus [\bot]\, \overline{n}\langle m \rhd \mathtt{RECV}\rangle\\
&\xrightarrow{\ell}_{ipc} [\bot \sqcap \mathtt{PERMS}']\, E\{m/x\}\\
&= [\bot]\, E\{m/x\}\\
&\triangleq \Gamma \cdot ([\mathtt{PERMS}']\, E\{m/x\}),
\end{aligned}$$

where the reduction step can be performed. In fact, $\mathtt{CALL} \sqsubseteq \mathtt{PERMS}$ by hypothesis (1) and $\mathtt{PERMS} = \bot$ by (24), i.e., $\mathtt{CALL} = \bot$ by anti-symmetry.

Assume then that $\Gamma \vdash_{\top} [\mathtt{PERMS}]\, \overline{n}\langle m \rhd \mathtt{RECV}\rangle : \tau \blacktriangleright \mathtt{P}$ was derived by (T-Call) after an application of (T-Perms), then we know that:

(25) $\Gamma \vdash n : \mathsf{Fun}(\mathtt{CALL}', \tau_n \rightarrow \tau_n')^{\mathtt{SECR}}$

(26) $\bot \sqsubset \mathtt{RECV} \sqcup \mathtt{SECR}$.

Since (18) and (25) hold, by Proposition 5.1.2 (Soundness of Secrecy Levels) we know that $\mathtt{SECR} = \bot$. Since $\mathtt{RECV} \sqsubseteq \mathtt{PERMS}'$ by hypothesis (2) and (20) holds, we know that $\mathtt{RECV} = \bot$ by anti-symmetry, thus $\mathtt{RECV} \sqcup \mathtt{SECR} = \bot$ and we get a contradiction by (26), i.e., the rule could not be applied and the case is trivial.

Finally, assume that $\Gamma \vdash_\top [\mathtt{PERMS}]\, \overline{n}\langle m \triangleright \mathtt{RECV}\rangle : \tau \blacktriangleright \mathtt{P}$ was derived by (T-Fail) after an application of (T-Perms), then we know that:

(27) $\Gamma \vdash n : \mathsf{Fun}(\mathtt{CALL}', \tau_n \to \tau_n')^{\mathtt{SECR}}$

(28) $\Gamma \vdash m : \tau_n''$

(29) $\mathtt{RECV} \sqcup \mathtt{SECR} = \bot \Rightarrow \mathcal{L}(\tau_n'') = \bot$

(30) $\tau = \mathsf{Un}$

(31) $\mathtt{P} = \mathtt{PERMS}$.

Since (18) and (27) hold, by Proposition 5.1.2 (Soundness of Secrecy Levels) we know that $\mathtt{SECR} = \bot$. Since $\mathtt{RECV} \sqsubseteq \mathtt{PERMS}'$ by hypothesis (2) and (20) holds, we know that $\mathtt{RECV} = \bot$ by anti-symmetry, thus $\mathtt{RECV} \sqcup \mathtt{SECR} = \bot$ and we get $\mathcal{L}(\tau_n'') = \bot$ by (29). This implies, using (28) and (T-Pub), that $\Gamma \vdash m : \mathsf{Un}$, hence by (19) and Lemma 5.1.4 (Substitution) we get $\Gamma \vdash_\bot [\mathtt{PERMS}']\, E\{m/x\} : \tau \blacktriangleright \bot$, hence we get $\Gamma \vdash_\top [\mathtt{PERMS}']\, E\{m/x\} : \tau \blacktriangleright \bot$ by Proposition 5.2.6 (Monotonicity of Typing). Now we note that:

$$\begin{aligned}
&\Gamma \cdot (\mathsf{def}\, n^\ell = \lambda(x \triangleleft \mathtt{CALL}).[\mathtt{PERMS}']\, E \setminus [\mathtt{PERMS}]\, \overline{n}\langle m \triangleright \mathtt{RECV}\rangle) \\
&= \Gamma \cdot (\mathsf{def}\, n^\ell = \lambda(x \triangleleft \mathtt{CALL}).[\bot]\, E \setminus [\mathtt{PERMS}]\, \overline{n}\langle m \triangleright \mathtt{RECV}\rangle) && \text{by (20)} \\
&\triangleq \mathsf{def}\, n^\ell = \lambda(x \triangleleft \mathtt{CALL}).[\bot]\, E \setminus [\mathtt{PERMS}]\, \overline{n}\langle m \triangleright \mathtt{RECV}\rangle && \text{by (31)} \\
&\xrightarrow{\ell}_{ipc} [\mathtt{PERMS} \sqcap \bot]\, E\{m/x\} \\
&= [\bot]\, E\{m/x\} \\
&\triangleq \Gamma \cdot ([\mathtt{PERMS}']\, E\{m/x\}),
\end{aligned}$$

where the reduction step can be performed, since $\mathtt{CALL} \sqsubseteq \mathtt{PERMS}$ by (1).

Case (R-Let): assume $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \xrightarrow{\alpha} \mathsf{let}\ x = E_1'\ \mathsf{in}\ E_2$ by the premise $E_1 \xrightarrow{\alpha} E_1'$. By hypothesis we know that $\Gamma \vdash_\top \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : \tau' \blacktriangleright \mathtt{P} \sqcup \mathtt{Q}$, which must follow by an instance of (T-Let). Hence, we have $\Gamma \vdash_\top E_1 : \tau \blacktriangleright \mathtt{P}$ and $\Gamma, x : \tau \vdash_\top E_2 : \tau' \blacktriangleright \mathtt{Q}$. By inductive hypothesis we have $\Gamma \vdash_\top E_1' : \tau \blacktriangleright \mathtt{P}'$ with $\mathtt{P}' \sqsubseteq \mathtt{P}$, hence $\Gamma \vdash_\top \mathsf{let}\ x = E_1'\ \mathsf{in}\ E_2 : \tau \blacktriangleright \mathtt{P}' \sqcup \mathtt{Q}$ by (T-Let).

Again by inductive hypothesis, we also know that $\Gamma \cdot E_1 \xrightarrow{\alpha}_{ipc} E_1''$ with $E_1'' \sqsupseteq \Gamma \cdot E_1'$, hence we have:

$$\begin{aligned}
\Gamma \cdot (\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2) &\triangleq \mathsf{let}\ x = (\Gamma \cdot E_1)\ \mathsf{in}\ ((\Gamma, x : \tau) \cdot E_2) \\
&\xrightarrow{\alpha}_{ipc} \mathsf{let}\ x = E_1''\ \mathsf{in}\ ((\Gamma, x : \tau) \cdot E_2) \\
&\sqsupseteq \mathsf{let}\ x = (\Gamma \cdot E_1')\ \mathsf{in}\ ((\Gamma, x : \tau) \cdot E_2) \\
&\triangleq \Gamma \cdot (\mathsf{let}\ x = E_1'\ \mathsf{in}\ E_2).
\end{aligned}$$

Case (R-Return): assume $\mathsf{let}\ x = [\mathtt{PERMS}]\, n\ \mathsf{in}\ E \dot{\rightarrow} E\{n/x\}$. By hypothesis we know that $\Gamma \vdash_{\top} \mathsf{let}\, x = [\mathtt{PERMS}]\, n\, \mathsf{in}\, E : \tau' \blacktriangleright \mathtt{P} \sqcup \mathtt{Q}$, which must follow by an instance of (T-Let). Hence, we have $\Gamma \vdash_{\top} [\mathtt{PERMS}]\, n : \tau \blacktriangleright \mathtt{P}$ and $\Gamma, x : \tau \vdash_{\top} E : \tau' \blacktriangleright \mathtt{Q}$ with $x \notin \mathit{dom}(\Gamma)$. The former judgement must have been derived by an application of (T-Val) after an instance of (T-Perms), thus we know that $\Gamma \vdash n : \tau$ and by Lemma 5.1.4 (Substitution) we have $\Gamma \vdash_{\top} E\{n/x\} : \tau' \blacktriangleright \mathtt{Q}$.

Now we note that:

$$\begin{aligned}\Gamma \cdot (\mathsf{let}\ x = [\mathtt{PERMS}]\, n\ \mathsf{in}\ E) &\triangleq \mathsf{let}\ x = [\mathtt{P} \sqcap \mathtt{PERMS}]\, n\ \mathsf{in}\ ((\Gamma, x : \tau) \cdot E)\\ &\dot{\rightarrow}_{\mathit{ipc}} ((\Gamma, x : \tau) \cdot E)\{n/x\}\\ &= \Gamma \cdot (E\{n/x\}).\end{aligned}$$

The last step uses Lemma 5.1.4 (Substitution) and some simple syntactic observations to conclude.

Case (R-Restr): assume $(\nu n)\, E \xrightarrow{\alpha} (\nu n)\, E'$ by the premise $E \xrightarrow{\alpha} E'$. By hypothesis we know that $\Gamma \vdash_{\top} (\nu n)\, E : \tau' \blacktriangleright \mathtt{PERMS}$, which must follow by an instance of (T-Restr), hence we have $\Gamma, n : \tau \vdash_{\top} E : \tau' \blacktriangleright \mathtt{PERMS}$ with $n \notin \mathit{dom}(\Gamma)$. By inductive hypothesis we have $\Gamma, n : \tau \vdash_{\top} E' : \tau' \blacktriangleright \mathtt{PERMS}'$ for some $\mathtt{PERMS}' \sqsubseteq \mathtt{PERMS}$, hence $\Gamma \vdash_{\top} (\nu n)\, E' : \tau' \blacktriangleright \mathtt{PERMS}'$ by (T-Restr).

Again by inductive hypothesis, we also know that $(\Gamma, n : \tau) \cdot E \xrightarrow{\alpha}_{\mathit{ipc}} E''$ with $E'' \sqsupseteq (\Gamma, n : \tau) \cdot E'$, hence we have:

$$\begin{aligned}\Gamma \cdot (\nu n)\, E &\triangleq (\nu n)\, ((\Gamma, n : \tau) \cdot E)\\ &\xrightarrow{\alpha}_{\mathit{ipc}} (\nu n)\, E''\\ &\sqsupseteq (\nu n)\, ((\Gamma, n : \tau) \cdot E')\\ &\triangleq \Gamma \cdot (\nu n)\, E'.\end{aligned}$$

Case (R-Eval): assume $D \setminus E \xrightarrow{\alpha} D \setminus E'$ by the premise $E \xrightarrow{\alpha} E'$. By hypothesis we know that $\Gamma \vdash_{\top} D \setminus E : \tau \blacktriangleright \mathtt{PERMS}$, which must follow by an instance of (T-Eval), hence we have $\Gamma \vdash D$ and $\Gamma \vdash_{\top} E : \tau \blacktriangleright \mathtt{PERMS}$. By inductive hypothesis we have $\Gamma \vdash_{\top} E' : \tau' \blacktriangleright \mathtt{PERMS}'$ for some $\mathtt{PERMS}' \sqsubseteq \mathtt{PERMS}$, hence $\Gamma \vdash_{\top} D \setminus E : \tau \blacktriangleright \mathtt{PERMS}'$ by (T-Store).

Again by inductive hypothesis, we also know that $\Gamma \cdot E \xrightarrow{\alpha}_{\mathit{ipc}} E''$ with $E'' \sqsupseteq \Gamma \cdot E'$, hence we have:

$$\begin{aligned}\Gamma \cdot (D \setminus E) &\triangleq D \setminus (\Gamma \cdot E)\\ &\xrightarrow{\alpha}_{\mathit{ipc}} D \setminus E''\\ &\sqsupseteq D \setminus (\Gamma \cdot E')\\ &\triangleq \Gamma \cdot (D \setminus E').\end{aligned}$$

Case (R-STRUCT): assume $E \xrightarrow{\alpha} E'$ by the premises $E \Rrightarrow E_1$, $E_1 \xrightarrow{\alpha} E_2$ and $E_2 \Rrightarrow E'$. By hypothesis we know that $\Gamma \vdash_{\top} E : \tau \blacktriangleright \texttt{PERMS}$, hence $\Gamma \vdash_{\top} E_1 : \tau \blacktriangleright \texttt{PERMS}$ by Lemma 5.1.5 (Heating Preserves Typing). By inductive hypothesis we then have $\Gamma \vdash_{\top} E_2 : \tau \blacktriangleright \texttt{PERMS}'$ for some $\texttt{PERMS}' \sqsubseteq \texttt{PERMS}$, hence we have $\Gamma \vdash_{\top} E' : \tau \blacktriangleright \texttt{PERMS}'$ again by Lemma 5.1.5 (Heating Preserves Typing).

Now we show the second part of the statement. Using Lemma 5.2.3 (Lowering Respects Heating), by $E \Rrightarrow E_1$ we have $\Gamma \cdot E \Rrightarrow E'' \sqsupseteq \Gamma \cdot E_1$ for some E''. By inductive hypothesis, we know that $\Gamma \cdot E_1 \xrightarrow{\alpha}_{ipc} E_1' \sqsupseteq \Gamma \cdot E_2$ for some E_1', hence by Lemma 5.2.5 (Monotonicity of Reduction) we get $E'' \xrightarrow{\alpha}_{ipc} E_1'' \sqsupseteq E_1' \sqsupseteq \Gamma \cdot E_2$ for some E_1''. Using again Lemma 5.2.3 (Lowering Respects Heating), by $E_2 \Rrightarrow E'$ we have $\Gamma \cdot E_2 \Rrightarrow E_2'' \sqsupseteq \Gamma \cdot E'$ for some E_2''. By Lemma 5.2.4 (Monotonicity of Heating) we then have $E_1'' \Rrightarrow E_2' \sqsupseteq E_2''$ for some E_2' and we conclude $\Gamma \cdot E \xrightarrow{\alpha}_{ipc} E_2' \sqsupseteq \Gamma \cdot E'$ by an application of (R-STRUCT).

□

5.3 Proof of (robust) safety

Proposition 5.3.1 (Equivalence up to Permissions). *The following statements hold:*

(i) for every pair of expressions E_1, E_2 such that $E_1 \sqsubseteq E_2$, we have $E_1 \asymp E_2$;

(ii) for every expression E such that $\Gamma \cdot E$ is defined, we have $\Gamma \cdot E \asymp E$.

Proof. Point (i) follows by induction on the derivation of $E_1 \sqsubseteq E_2$, while point (ii) follows by Proposition 5.2.2 (Soundness of Lowering) and point (i). □

Restatement of Theorem 4.5.1. If $\Gamma \vdash_{\top} E : \tau \blacktriangleright \texttt{P}$, then $E \preccurlyeq E$.

Proof. Let $\mathcal{R} = \{(E_1, E_2) \mid \Gamma \vdash_{\top} E_1 : \tau \blacktriangleright \texttt{P}_1 \wedge E_2 \sqsupseteq \Gamma \cdot E_1\}$. We show that $\mathcal{R}$ is a simulation. Notice first that for every E_1, E_2 such that $(E_1, E_2) \in \mathcal{R}$ we have $E_1 \asymp E_2$ by Proposition 5.3.1 (Equivalence up to Permissions) and the transitivity of the $\asymp$ relation, hence we just need to show that the transitions match as prescribed.

Let $(E_1, E_2) \in \mathcal{R}$, then we know that $\Gamma \vdash_{\top} E_1 : \tau \blacktriangleright \texttt{P}_1$ and $E_2 \sqsupseteq \Gamma \cdot E_1$. Assume $E_1 \xrightarrow{\alpha} E_1'$, then by Theorem 5.2.7 (Simulation-Aware Subject Reduction) we have $\Gamma \vdash_{\top} E_1' : \tau \blacktriangleright \texttt{P}_1'$ with $\texttt{P}_1' \sqsubseteq \texttt{P}_1$ and $\Gamma \cdot E_1 \xrightarrow{\alpha}_{ipc} E_1''$ for some $E_1'' \sqsupseteq \Gamma \cdot E_1'$. By Lemma 5.2.5 (Monotonicity of Reduction) we then have $E_2 \xrightarrow{\alpha}_{ipc} E_2'$ for some $E_2' \sqsupseteq E_1'' \sqsupseteq \Gamma \cdot E_1'$, hence $(E_1', E_2') \in \mathcal{R}$ and we conclude that $\mathcal{R}$ is a simulation.

Finally, we note that by Proposition 5.2.2 (Soundness of Lowering) we have $E \sqsupseteq \Gamma \cdot E$, hence $(E, E) \in \mathcal{R}$ and we conclude $E \preccurlyeq E$ as desired. □

Restatement of Lemma 4.5.2. Let O be an opponent and let $\Gamma \vdash u : \mathsf{Un}$ for all $u \in \mathit{fnfv}(O)$, then $\Gamma \vdash O$.

Proof. Let E be any expression such that each permission assignment occurring within E is $\bot$. Since the structure of definitions and expressions is given by mutually inductive productions, we simultaneously prove the following statements:

(i) $\forall u \in \mathit{fnfv}(E) : \Gamma \vdash u : \mathsf{Un} \Rightarrow \Gamma \vdash_{\bot} E : \mathsf{Un} \blacktriangleright \bot$

(ii) $\forall u \in \mathit{fnfv}(O) : \Gamma \vdash u : \mathsf{Un} \Rightarrow \Gamma \vdash O$.

The proof of point (i) is by induction on the structure of E, while the proof of point (ii) is by induction on the structure of O. $\square$

Restatement of Theorem 4.5.3. Let $\mathcal{L}(\tau) = \bot$ for every u such that $\Gamma(u) = \tau$. If $\Gamma \vdash_{\top} E : \tau \blacktriangleright \mathtt{P}$, then E is robustly safe against privilege escalation attacks.

Proof. Let O be an arbitrary opponent. Let Γ^* be the typing environment defined as follows:

$$\Gamma^*(u) = \begin{cases} \Gamma(u) & \text{if } u \in \mathit{dom}(\Gamma) \\ \mathsf{Un} & \text{if } u \notin \mathit{dom}(\Gamma) \wedge u \in \mathit{fnfv}(O) \end{cases}$$

We let $\Gamma^*(u)$ be undefined for any u such that $u \notin \mathit{dom}(\Gamma) \cup \mathit{fnfv}(O)$.

Now we note that $\forall u \in \mathit{dom}(\Gamma^*) : \Gamma^* \vdash u : \mathsf{Un}$, hence $\Gamma^* \vdash O$ by Lemma 4.5.2 (Opponent Typability). By Lemma 5.1.3 (Weakening) we also have $\Gamma^* \vdash_{\top} E : \tau \blacktriangleright \mathtt{P}$, thus $\Gamma^* \vdash_{\top} O \setminus E : \tau \blacktriangleright \mathtt{P}$ by rule (T-EVAL). Hence, the conclusion follows by Theorem 4.5.1 (Type Safety).

$\square$

Conclusions

Authorization policies are a well-understood, successful and widespread security mechanism, whose applicability and generality have been proved in a number of different settings. Despite this popularity and a long-standing research tradition, assessing the effectiveness of a given authorization policy is still a challenging problem: first, the actual security implications underlying the policy must be clearly identified and understood, which already is a non-trivial task, since the policy specification and the intended security requirements are typically expressed at very different levels; second, proving that a system or an implementation comply with a specific authorization policy requires significant domain-specific expertise and careful reasoning about the attacker model.

The present thesis contributes to the research line of policy verification with the first formal semantics for `grsecurity`, a fundamental building block for any rigorous verification procedure for the role-based access control system of `grsecurity`. The thesis addresses also the problem of policy enforcement, which is tackled specifically on application code: for F# applications, a more expressive policy language is considered with respect to the original proposal in [10]; for Android applications, the problem of statically enforcing a more robust access control policy is addressed.

A thorough validation of the presented analysis techniques on existing codebases is an important research task, which will require substantial engineering effort in the next future. At the time of writing, the importance of integrating the analysis performed by `gran` with information from the underlying file system has been recognised as a feature of fundamental importance to simplify the process of policy verification and make the analysis more precise. Additionally, we are carrying out several case studies on Android security through the usage of `Lintent`.

Bibliography

[1] man page for function `setreuid`.

[2] Smali: An assembler/disassembler for android's dex format. `http://code.google.com/p/smali/`.

[3] Sponsor page of `grsecurity`.

[4] Martín Abadi and Bruno Blanchet. Secrecy types for asymmetric communication. *Theor. Comput. Sci.*, 3(298):387–415, 2003.

[5] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proc. 28th Symposium on Principles of Programming Languages (POPL)*, pages 104–115. ACM, 2001.

[6] Alessandro Armando, Gabriele Costa, and Alessio Merlo. Formal modeling and verification of the android security framework. In *TGC2012*, pages xx–xx, 2012. To Appear.

[7] Alessandro Armando and Silvio Ranise. Automated symbolic analysis of arbac-policies. In Jorge Cuéllar, Javier Lopez, Gilles Barthe, and Alexander Pretschner, editors, *STM*, volume 6710 of *Lecture Notes in Computer Science*, pages 17–34. Springer, 2010.

[8] Michael Backes, Catalin Hriţcu, and Matteo Maffei. Union and Intersection Types for Secure Protocol Implementations. In *Proc. Theory of Security and Applications (TOSCA)*, Lecture Notes in Computer Science, pages 1–28. Springer-Verlag, 2011.

[9] David E. Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical report, MITRE Corporation, 1973.

[10] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2), 2011.

[11] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic Protocol Synthesis and Verification for Multiparty Sessions. In *Proc. 22nd IEEE Symposium on Computer Security Foundations (CSF)*, pages 124–140. IEEE Computer Society Press, 2009.

[12] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Modular Verification of Security Protocol Code by Typing. In *Proc. 37th Symposium on Principles of Programming Languages (POPL)*, pages 445–456. ACM Press, 2010.

[13] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified Interoperable Implementations of Security Protocols. *ACM Transactions on Programming Languages and Systems*, 31(1), 2008.

[14] Kenneth J. Biba. Integrity Considerations for Secure Computer Systems. Technical report, USAF Electronic Systems Division, 1977.

[15] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proc. 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA)*, pages 301–320. ACM Press, 2007.

[16] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *CSFW'01*, pages 82–96. IEEE, 2001.

[17] Chiara Braghin, Daniele Gorla, and Vladimiro Sassone. A distributed calculus for role-based access control. In *CSFW*, pages 48–60, 2004.

[18] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS*, 2012. To appear.

[19] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. Resource-Aware Authorization Policies for Statically Typed Cryptographic Protocols. In *Proc. 24th IEEE Symposium on Computer Security Foundations (CSF)*, pages 83–98. IEEE Computer Society Press, 2011.

[20] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. Affine Refinement Types for Authentication and Authorization. In *Proc. 7th Symposium on Trustworthly Global Computing (TGC)*, 2012.

[21] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. Logical Foundations of Secure Resource Management in Protocol Implementations. In *Proc. 2nd International Conference on Principles of Security and Trust (POST)*, 2012. To appear.

[22] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Marco Squarcina. Gran: Model checking grsecurity rbac policies. In Stephen Chong, editor, *CSF*, pages 126–138. IEEE, 2012.

[23] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Dynamic Types for Authentication. *Journal of Computer Security*, 15(6):563–617, 2007.

[24] Peter C. Chapin, Christian Skalka, and Xiaoyang Sean Wang. Authorization in Trust Management: Features and Foundations. *ACM Computing Surveys*, 40(3), 2008.

[25] Avik Chaudhuri. Language-based security on Android. In *PLAS*, pages 1–7, 2009.

[26] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *MobiSys*, pages 239–252, 2011.

[27] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[28] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[29] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *ISC*, pages 346–360, 2010.

[30] Nicholaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[31] William Enck. Defending users against smartphone apps: Techniques and future directions. In *ICISS*, pages 49–70, 2011.

[32] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.

[33] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of Android application security. In *USENIX Security Symposium*, 2011.

[34] William Enck, Machigar Ongtang, and Patrick Drew McDaniel. Understanding android security. *IEEE Security & Privacy*, 7(1):50–57, 2009.

[35] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Stowaway - android permissions demystified. `http://www.android-permissions.org/`.

[36] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *ACM Conference on Computer and Communications Security*, pages 627–638, 2011.

[37] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[38] Riccardo Focardi and Matteo Maffei. Types for Security Protocols. Technical Report CS-2010-3, University of Venice, 2010. Available at `http://www.lbs.cs.uni-saarland.de/resources/types-security.pdf`.

[39] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A Type Discipline for Authorization Policies. In *ESOP'05*, LNCS, pages 141–156. Springer, 2005.

[40] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A Type Discipline for Authorization in Distributed Systems. In *CSF'07*, pages 31–45. IEEE, 2007.

[41] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular Code-Based Cryptographic Verification. In *Proc. 18th ACM Conference on Computer and Communications Security (CCS)*, pages 341–350. ACM Press, 2011.

[42] Michael Fox, John Giordano, Lori Stotler, and Arun Thomas. SELinux and grsecurity: A case study comparing linux security kernel enhancements. University of Virginia.

[43] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and enhancing Android's permission system. In *ESORICS*, volume 7459 of *Lecture Notes in Computer Science*, pages 1–18, 2012. To appear.

[44] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications, 2009. Technical report, University of Maryland.

[45] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for java bytecode. In *SAS*, pages 199–219, 2000.

[46] Jean-Yves Girard. Linear Logic: Its Syntax and Semantics. In *Advances in Linear Logic*, volume 22 of *London Mathematical Society LNS*, pages 1–42. Cambridge University Press, 1995.

[47] Mikhail I. Gofman, Ruiqi Luo, Ayla C. Solomon, Yingbin Zhang, Ping Yang, and Scott D. Stoller. Rbac-pat: A policy analysis tool for role based access control. In Stefan Kowalewski and Anna Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 46–49. Springer, 2009.

[48] Google Inc. Reference documentation for `android.app.PendingIntent`. `http://developer.android.com/reference/android/app/PendingIntent.html`.

[49] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. *Electr. Notes Theor. Comput. Sci.*, 16(3):248–264, 1998.

[50] Andrew D. Gordon and Alan Jeffrey. Authenticity by Typing for Security Protocols. *JCS*, 11(4):451–519, 2003.

[51] Andrew D. Gordon and Alan Jeffrey. Types and Effects for Asymmetric Cryptographic Protocols. *JCS*, 12(3):435–484, 2004.

[52] Joshua D. Guttman, F. Javier Thayer, Jay A. Carlson, Jonathan C. Herzog, John D. Ramsdell, and Brian T. Sniffen. Trust Management in Strand Spaces: A Rely-Guarantee Method. In *Proc. 13th European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science, pages 325–339. Springer-Verlag, 2004.

[53] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, 1976.

[54] Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. Lambda-rbac: Programming with role-based access control. *Logical Methods in Computer Science*, 4(1), 2008.

[55] Karthick Jayaraman, Vijay Ganesh, Mahesh V. Tripunitara, Martin C. Rinard, and Steve J. Chapin. Automatic error finding in access-control policies. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 163–174. ACM, 2011.

[56] Somesh Jha, Ninghui Li, Mahesh V. Tripunitara, Qihua Wang, and William H. Winsborough. Towards formal verification of role-based access control policies. *IEEE Trans. Dependable Sec. Comput.*, 5(4):242–255, 2008.

[57] Ninghui Li and Mahesh V. Tripunitara. Security analysis in role-based access control. *ACM Trans. Inf. Syst. Secur.*, 9(4):391–420, 2006.

[58] Amiya Kumar Maji, Fahad A. Arshad, Saurabh Bagchi, and Jan S. Rellermeyer. An empirical study of the robustness of inter-component communication in android. In *DSN*, pages 1–12, 2012.

[59] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *ICFP'03*, pages 213–225. ACM Press, 2003.

[60] Robin Milner. Functions as Processes. *MSCS*, 2(2):119–141, 1992.

[61] James H. Morris. Protection in Programming Languages. *Communications of the ACM*, 16(1):15–21, 1973.

[62] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.

[63] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A Type System for Borrowing Permissions. In *POPL'12*, pages 557–570. ACM Press, 2012.

[64] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[65] Project Lombok. Reference documentation for `lombok.javac` abstract syntax tree. `http://projectlombok.org/api/lombok/javac/package-summary.html`.

[66] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21, 2003.

[67] Ravi S. Sandhu, Venkata Bhamidipati, and Qamar Munawer. The arbac97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur.*, 2(1):105–135, 1999.

[68] Amit Sasturkar, Ping Yang, Scott D. Stoller, and C. R. Ramakrishnan. Policy analysis for administrative role based access control. In *CSFW*, pages 124–138. IEEE Computer Society, 2006.

[69] Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. A formal model to analyze the permission authorization and enforcement in the android framework. In *SocialCom/PASSAT*, pages 944–951, 2010.

[70] Alvise Spanò. Information extraction from weakly-typed code by typing. Ph.D. Thesis (Under review), Università Ca' Foscari Venezia.

[71] Brad Spengler. Increasing performance and granularity in role-based access control systems, 2004.

[72] Brad Spengler. Changelog of `grsecurity`, February 2012. commit 3981059c35e8463002517935c28f3d74b8e3703c.

[73] Brad Spengler. Private communication, February 2012.

[74] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *USENIX'88*, pages 191–202. USENIX Association, 1988.

[75] Scott D. Stoller, Ping Yang, C. R. Ramakrishnan, and Mikhail I. Gofman. Efficient policy analysis for administrative role based access control. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 445–455. ACM, 2007.

[76] Eijiro Sumii and Benjamin C. Pierce. A Bisimulation for Dynamic Sealing. *Theoretical Computer Science*, 375(1-3):169–192, 2007.

[77] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-Class State Change in Plaid. In *OOPSLA'11*, pages 713–732. ACM Press, 2011.

[78] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure Distributed Programming with Value-Dependent Types. In *Proc. 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 266–278. ACM Press, 2011.

[79] Alwen Tiu and Alberto Momigliano. Induction and co-induction in sequent calculus. *CoRR*, abs/0812.4727, 2008.

[80] Naoyuki Tomura. llprover - A Linear Logic Prover. `http://bach.istc.kobe-u.ac.jp/llprover/`.

[81] Jesse Tov and Riccardo Pucella. Stateful Contracts for Affine Types. In *Proc. 19th European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science, pages 550–569. Springer-Verlag, 2010.

[82] Anne S. Troelstra. Lectures on Linear Logic. CSLI Stanford, LNS, vol. 29, 1992.

[83] Nan Zhang, Mark Ryan, and Dimitar P. Guelev. Synthesising verified access control systems through model checking. *Journal of Computer Security*, 16(1):1–61, 2008.