Universität des Saarlandes
Fakultät für Mathematik und Informatik (MI)
(ehemals Naturwissenschaftlich-Technische Fakultät I)

A Theory of Types for Security and Privacy

Fabienne Sophie Eigner

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

Saarbrücken, 2015

Tag des Kolloquiums:	12. Dezember 2016
Dekan:	Prof. Dr. Frank-Olaf Schreyer

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Bernd Finkbeiner
Berichterstattende:	Prof. Dr. Matteo Maffei
	Prof. Dr. Michael Backes
	Prof. Dr. Véronique Cortier
Akademischer Mitarbeiter:	Dr. Robert Künnemann

Zusammenfassung

Im modernen Internet sind kryptographische Protokolle allgegenwärtig. Ihre Entwicklung ist jedoch schwierig und eine manuelle Sicherheitsanalyse mühsam und fehleranfällig. Ein Mangel an exakten Sicherheitsbeweisen führt daher zu oft gravierenden Sicherheitsmängeln in vielen Protokollen.

Um Datenschutz und Sicherheit kryptographischer Protokolle zu verbessern und deren Verifikation zu vereinfachen, konzentriert sich ein Großteil der Forschung auf formale Protokollanalyse. Dies führte zur Entwicklung automatischer Tools, die auf symbolischen Kryptographie-Abstraktionen basieren. Jedoch gibt es weiterhin zahlreiche Protokolle und Sicherheitseigenschaften, deren Analyse zu komplex für aktuelle Systeme ist.

Diese Dissertation stellt drei neuartige Frameworks zur Verifikation von Sicherheitsprotokollen und ihren Implementierungen vor. Sie nutzen eine leistungsstarker Typisierung für Sicherheit und Datenschutz und verbessern damit die aktuelle, Beschränkungen unterworfene Situation. Mit AF7 präsentieren wir die erste statische Typisierung von Protokollimplementierungen bezüglich Sicherheitseigenschaften, die in affiner Logik formuliert sind. Zudem sorgt unsere neuartige typbasierte, automatische Analysetechnik von elektronischen Wahlsystemen für Datenschutz und Überprüfbarkeit im Wahlprozess. Schließlich stellen wir mit DF7 das erste affine Typsystem zur statischen, automatischen Verifikation der sogenannten Distributed Differential Privacy in Protokollimplementierungen vor.

Abstract

Cryptographic protocols are ubiquitous in the modern web. However, they are notoriously difficult to design and their manual security analysis is both tedious and error-prone. Due to the lack of rigorous security proofs, many protocols have been discovered to be flawed.

To improve the security and privacy guarantees of cryptographic protocols and their implementations and to facilitate their verification, a lot of research has been directed towards the formal analysis of such protocols. This has led to the development of several automated tools based on symbolic abstractions of cryptography. Unfortunately, there are still various cryptographic protocols and properties that are out of the scope of current systems.

This thesis introduces three novel frameworks for the verification of security protocols and their implementations based on powerful types for security and privacy, overcoming the limitations of current state-of-the-art approaches. With AF7 we present the first type system that statically enforces the safety of cryptographic protocol implementations with respect to authorization policies expressed in *affine logic*. Furthermore, our novel approach for the automated analysis of e-voting systems based on refinement type systems can be used to enforce both *privacy* and *verifiability*. Finally, with DF7, we present the first affine, distance-aware type system to statically and automatically enforce *distributed differential privacy* in cryptographic protocol implementations.

Preface

The following thesis is based on several published research papers to which I contributed as a main author during my Ph.D. studies in Computer Science at Saarland University, Germany.

Chapter 2 presents the results of a long-standing research collaboration with Michele Bugliesi[*], Stefano Calzavara[*], and Matteo Maffei[†] on resource-aware authorization policies. Preliminary results were presented at the 24th IEEE Computer Security Foundations Symposium [1] in 2011 and at the 7th International Symposium on Trustworthy Global Computing [2] in 2012. This chapter mainly focusses on the most recent and comprehensive results of this research project that significantly extend the previous work. These results were presented at the 2nd Conference on Principles of Security and Trust [3] in 2013, where they have been awarded the *EATCS Award* for the best theory paper at ETAPS. A comprehensive journal version [4] was published in ACM Transactions on Programming Languages and Systems in August 2015.

Chapter 3 results from an international research collaboration with Véronique Cortier[‡], Steve Kremer[‡], Matteo Maffei[*], and Cyrille Wiedling[§] on type-based verification of electronic voting systems. The resulting paper was presented at the 4th Conference on Principles of Security and Trust [5] in 2015. The corresponding technical report [6] was published in the IACR Cryptology ePrint Archive.

Chapter 4 is the outcome of a joint project with Matteo Maffei[†] on distributed differential privacy. The results were presented at the 26th IEEE Computer Security Foundations Symposium [7] in 2013.

In addition to the above papers closely related to this dissertation, I was also strongly involved in the development of PrivaDA, a generic framework for privacy-preserving data aggregation with optimal utility. The framework is the result of a collaboration with Aniket Kate[†], Matteo Maffei[†], Francesca Pampaloni[¶], and Ivan Pryvalov[†]. This work was accepted at the 30th Annual Computer Security Applications Conference [8] in 2014. The corresponding technical report [9] was published in the IACR Cryptology ePrint Archive and a revised version was published as a book chapter [10].

[*] Università Ca' Foscari Venezia, Italy

[†] Saarland University and CISPA, Germany

[‡] LORIA, CNRS & INRIA & University of Lorraine, France

[§] Université Catholique de Louvain, Belgium

[¶] formerly IMT Lucca, Italy

Acknowledgments

First and foremost, I would like to express my deep gratitude to Matteo Maffei, whose enthusiasm for security and formal methods and whose optimism, trust, and support were invaluable for the completion of this thesis.

In addition, I would like to thank Michael Backes for agreeing to review this dissertation and for introducing me to the world of cryptography and security many years ago.

There are many wonderful researchers that I had the pleasure of collaborating with during my PhD studies and whom I would like to thank for their enthusiasm and the fruitful discussions. In particular, I would like to thank Michele Bugliesi and Stefano Calzavara for our long and strong collaboration on affine type-checking, Véronique Cortier, Steve Kremer, and Cyrille Wiedling for their expertise on electronic voting, as well as Aniket Kate, Francesca Pampaloni, and Ivan Pryvalov for making the journey into the world of secure multi-party computations such an interesting one. A special thanks goes to Stefano for endless Skype sessions full of proofs, laughter, desperation, and never-wavering optimism.

I would like to thank my colleagues at the Secure and Privacy-preserving Systems group and CISPA for creating a nice and fun work environment and for lots of great discussions. In particular, I would like to thank Kim Pecina, for starting this journey with me, back when our research group was just tiny. Furthermore, my thanks go to our administrative staff who always lent a friendly ear whenever needed.

I am grateful for the love and acceptance of my friends and family, who always showed me that there *is* a life outside of work and I thank Fanie for making this life so much brighter.

Contents

List of Tables

List of Figures

Part I

Introduction

1
Introduction

Cryptographic protocols are ubiquitous, their applications range from exchanging keys and establishing secure communication channels to complex protocols, e.g., for e-commerce, e-payments, targeted advertising, electronic voting, and secure multi-party computations that can be used to let multiple mutually distrusting parties jointly and safely compute a result. However, security protocols and their implementations are notoriously difficult to design and their manual security analysis is both hard and prone to errors. In fact, security flaws have been discovered for both early academic protocols like Needham-Schroeder [12] and carefully designed de facto standards like SSL [13], PKCS #11 [14], and the SAML-based Single Sign-On for Google Apps [15].

To improve the security and privacy guarantees of cryptographic protocols and their implementations and to facilitate their verification a substantial research effort has been directed towards the formal analysis of cryptographic protocols, which has led to the development of several automated tools based on symbolic abstractions of cryptography.

1.1 Protocol analysis: state-of-the-art

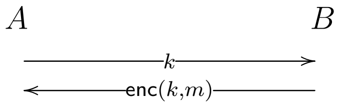
The majority of tools for the automated analysis of cryptographic protocols builds on one of the following two approaches.

1.1.1 Automated theorem provers

One line of research has focused on *automated theorem provers*, which build on a term-based abstraction of cryptography. Examples include ProVerif [16] and the CASPA [17] and Scyther [18] tools. Such approaches have been used successfully for enforcing various trace properties [16–19] and observational equivalence relations [20–22].

We will exemplify the intuition behind automated theorem provers for protocol verification on the example of the ProVerif tool. Intuitively, in ProVerif, a protocol is represented by a set of Horn clauses, i.e. logical formulas that are either simple logical predicates of the form $p(M_1, \ldots, M_n)$ (called *facts*) or logical implications of the form $F_1 \wedge \ldots \wedge F_n \Rightarrow F$ (called *Horn clauses*), where $F, F_1, \ldots, F_n$ are facts. Though several predicates p can be used, we will focus on the predicate $\mathsf{attacker}(M)$, which expresses the fact that the message M might be known to the attacker. The message M (also referred to as a *term* M) can either be a variable x, a name a representing an atomic value such as a key or a nonce (i.e., a random number), or the application of a function f. Names can be created by all principals and are fresh for all runs of the protocols. Typically, functions f model cryptographic or arithmetic operations such as encryption, hashing, or signatures. There exist two kinds of functions: constructors and destructors. Constructors, e.g., the encryption function enc, appear explicitly in the terms that represent messages. Destructors manipulate terms and are defined by a set of rewrite rules that model their symbolic meaning. For instance, decryption can be represented by the destructor dec, which is defined by the following rewrite rule: $\mathsf{dec}(M_1, \mathsf{enc}(M_1, M_2)) \to M_2$. This rewrite rule expresses that decrypting a ciphertext encrypted with the same secret key returns the cleartext.

Consider the following easy, but fundamentally flawed, key exchange protocol:



Alice first outputs the secret key k that she wants to share with Bob on the network in plain. Bob then uses k to encrypt the secret message m. Here, m and k are both fresh names, and the function enc is applied to the pair (k, m) to denote the encryption of m with key k.

Obviously, this protocol provides no secrecy at all: by outputting an allegedly secret key on the network, the attacker is given everything he needs to know to decrypt all messages encrypted with that key. While this flaw can be easily detected on such a small protocol by manual inspection, we will use this as a toy example to demonstrate how one can use ProVerif to formally verify the security of this and larger protocol. In this example, we will employ ProVerif for checking the secrecy of the key k and of the message m. This translates to checking whether $\mathsf{attacker}(k)$ and $\mathsf{attacker}(m)$ are satisfiable by the set of Horn clauses representing the protocol, symbolic cryptography, and attacker capabilities, i.e. whether the attacker can learn k and/or m from the message exchange. Here, the Dolev-Yao style attacker is assumed to be able to intercept all messages sent on the network, to synthesize new messages from the received ones, and to send all messages under his control. We will now exemplify some of the encoding of the protocol and the computation capabilities of the attacker as set of (slightly simplified) Horn clauses to show one possible attack the tool discovers. During his computations, an attacker can apply all functions to the messages he knows.

In our example, this is for instance modeled by the Horn clause

$$\mathsf{attacker}(x_1) \wedge \mathsf{attacker}(x_2) \Rightarrow \mathsf{attacker}(\mathsf{enc}(x_1, x_2)).$$

To allow an attacker the application of destructors, the rewrite rules are encoded like this one for decryption ($\mathsf{dec}(M_1, \mathsf{enc}(M_1, M_2)) \to M_2$):

$$\mathsf{attacker}(M_1) \wedge \mathsf{attacker}(\mathsf{enc}(M_1, M_2)) \Rightarrow \mathsf{attacker}(M_2)$$

The two protocol steps are modeled by the following two clauses: step 1 is modeled as

$$\mathsf{attacker}(k)$$

(since the key is simply output on the network in plain) and step 2 is modeled as

$$\mathsf{attacker}(k) \Rightarrow \mathsf{attacker}(\mathsf{enc}(k, m)).$$

The latter expresses the fact that Bob only sends his message after receiving the key that he uses for encryption and that his output message is then known to the attacker. From step 1 we can immediately deduce that the key k is known to the attacker (and thus not secret). Using this knowledge and step 2 together with the destructor rule for dec enables us to derive that $\mathsf{attacker}(m)$ is also satisfiable, i.e., m is not kept secret. ProVerif would thus output an attack and reject the protocol.

As mentioned before, ProVerif can additionally be used for verifying various trace properties and observational equivalence (for protocols with the same control-flow) following similar techniques. Unfortunately, there exist sophisticated cryptographic primitives (e.g., homomorphic encryption) and properties (e.g., differential privacy and some electronic voting properties like verifiability and coercion-resistance) that are currently not supported by ProVerif and the other automated tools mentioned above.

1.1.2 Type systems

Another line of research has focused on the design of *type systems* for cryptographic protocol analysis. Type systems are well-known tools from the area of programming languages. They are used to statically enforce the safety of a program and avoid running a faulty program, which could lead to a run that will crash. For instance, the typing rule

$$\frac{\Gamma \vdash M : \mathsf{int} \qquad \Gamma \vdash N : \mathsf{int}}{\Gamma \vdash M + N : \mathsf{int}}\ \textsc{Add}$$

states that two values M and N can be added together safely under the *typing environment* Γ that binds variables to their types if both M and N are integers. Their sum $M + N$ will then also be given type int. Assuming that there are no further typing rules for the syntactic case of addition, this rule ensures that in a well-typed program all additions will succeed at run-time.

For example, the program $x+4$ will type-check under the environment $\Gamma = x : \mathsf{int}$, which specifies that the variable x is used to denote an integer. In contrast, the program `'hello'`$+4$ will not type-check under any environment, since adding a string `'hello'` to an integer is forbidden by the typing rules, as it makes no semantic sense and such a program would crash at run-time.

Type systems for security and privacy build on a similar approach. As a very simple example, consider a type system that checks whether a message is allowed to be output on a certain channel or not. For example, posting a public picture on a public Facebook page should be OK, whereas posting a user's credit card information on such a site should be prevented, while sending this information over a trusted and secure connection to an online store should be fine as well. Consider the following simple typing rule:

$$\frac{\Gamma \vdash C : \mathsf{chan}(T) \qquad \Gamma \vdash M : T}{\Gamma \vdash \mathit{output}(C, M) : \mathsf{ok}}\ \textsc{Output}.$$

It specifies that an output of message M on channel C type-checks with type ok if the message M has some type T and the channel is of the corresponding channel type $\mathsf{chan}(T)$.

We assume the following typing environment Γ that binds both messages and channels to the following types that correspond to our intuition described above:

$$\Gamma = \mathit{facebook} : \mathsf{chan}(\mathsf{public}), \mathit{ssl2store} : \mathsf{chan}(\mathsf{private}), \mathit{pic1} : \mathsf{public}, \mathit{card} : \mathsf{private}.$$

Given the above typing rule, it can be easily seen that the first example protocol $\mathit{output}(\mathit{facebook}, \mathit{pic1})$ type-checks under Γ, while the second example protocol $\mathit{output}(\mathit{facebook}, \mathit{card})$ does not. Outputting credit card information on a secure connection, i.e., the protocol $\mathit{output}(\mathit{ssl2store}, \mathit{card})$, type-checks as well. Thus, a system with the above typing rule can be used to enforce the privacy of sensitive resources and to statically prevent data leakage.

Type systems, in particular, refinement type systems that allow for tracking pre- and post-conditions on security-sensitive code, have been used successfully for enforcing various trace properties such as authentication [23–29], and classical authorization policies [30–33]. Furthermore, they also proved capable to enforce even observational equivalence relations, such as secrecy [34] and relational properties [35].

1.1.3 Comparing the two approaches

On the one hand, type systems are to some extent less precise than theorem provers and are not suitable to automatically report attacks, in that (contrary to ProVerif for example) they do not explicitly compute abstractions of execution traces, though investigating why a certain protocol does not type-check can often lead to the discovery of previously overlooked attacks.

On the other hand, one central advantage of type system is their inherent modularity, e.g., the code of multiple participants can be checked separately (even in parallel) and the security of the overall protocol then follows by compositionality. Therefore, this approach scales better to large-scale protocols. The modularity of type-checking can be seen by comparing the two examples in Section 1.1.1 and Section 1.1.2. In Section 1.1.1, the analysis of ProVerif needs to take the whole protocol, i.e., the code of both participants Alice and Bob, into account. This is necessary to generate the complete set of attacker knowledge in order to deduce the secrecy of message m by checking the satisfiability of the predicate $\mathsf{attacker}(m)$. When we imagine to type-check the same protocol using intuitions similar to the ones in Section 1.1.2, it is clear that type-checking only the code of Bob individually will already reveal the flaw: the key k received on the public network will automatically be assigned a *public type* and will thus not be available for encrypting a *private message*. Hence, the code of Bob (and by compositionality the overall protocol) will not type-check.

Furthermore, as we mentioned before, type systems enable reasoning about sophisticated cryptographic schemes [31–34].

1.2 Limitations of existing approaches.

Although type systems look promising and seem to be an optimal choice for the automated verification of cryptographic protocol implementations, previous type systems, though powerful, fall short in offering support to verify a variety of security and privacy properties, either due to them being out of the scope of the respective systems or due to the fact that it is unknown whether, and how, it is possible to enforce them using a type-based approach.

In this thesis, we aim at closing this gap by developing and using type systems to address the following three classes of properties and protocols that were out of the scope of previous approaches (type-based or other).

Resource-aware authorization policies. Authorization policies provide a well-established device for the security analysis and specification of distributed protocols and applications. Given an access request to a sensitive resource in a system, an authorization policy (expressed in some authorization logic) determines whether the request should be allowed.

For instance, the authorization policy defined by the formula

$$\mathsf{Req}(C, S, m) \wedge \mathsf{Paid}(C, S, m) \wedge \mathsf{Available}(m) \Rightarrow \mathsf{Grant}(C, S, m)$$

can be used to express the property that a client C of an online streaming service S is only granted permission to watch a requested movie m if that movie is available and the client paid for it.

However, this policy allows the client to watch each movie arbitrarily often, which falls short in capturing many real-life scenarios in which the streaming service might want to ensure that paying once does not grant infinite access to a resource. Such a more restrictive *resource-aware* property can be expressed using *affine logic*, which treats affine formulas as resources that can be used only once in a proof derivation. The corresponding policy expressed in affine logic looks as follows:

$$!(\mathsf{Req}(C, S, m) \otimes \mathsf{Paid}(C, S, m) \otimes \mathsf{Available}(m) \multimap \mathsf{Grant}(C, S, m)).$$

Here, $\otimes$ represents affine, i.e., resource-consuming, conjunction, $\multimap$ denotes affine implication, and the replication operator ! expresses the fact that the overall policy itself is not affine but instead holds arbitrarily often. According to this policy, each payment predicate $\mathsf{Paid}(C, S, m)$ will only justify one permission $\mathsf{Grant}(C, S, m)$.

Type systems can be used to prove that a cryptographic protocol complies with a given authorization policy. Previous type systems, however, are often parametric in their choice of underlying authorization logic but do *not* allow for the verification of authorization policies expressed in affine logic.

Electronic voting. *Electronic voting* promises a lot of advantages over traditional voting: it is not only fast and convenient to use (e.g., in the case of *remote* electronic voting, it can be used from the comfort of home, using a personal device such as a laptop or smartphone), but it also features additional security properties that cannot be achieved with traditional voting. Typical examples of such properties are individual or universal verifiability of correctness, i.e. the chance to verify that one's own vote has been included in the election result, and that the complete election result has been calculated correctly.

To achieve such properties, electronic voting protocols commonly employ advanced cryptographic primitives. This makes their design as well as a rigorous security and privacy analysis quite challenging. As a matter of fact, existing automated analysis techniques, which are mostly based on automated theorem provers, are inadequate to deal with commonly used cryptographic primitives, such as homomorphic encryption, mix-nets, and zero-knowledge proofs, as well as some fundamental security properties, such as verifiability.

Distributed differential privacy. *'How many people in Germany suffer from diabetes? How many of them are members of your insurance?'* Such statistical information about data collected in databases is often released to the public. On the one hand, disclosing this kind of information is often desirable for analyzing trends, performing marketing studies, or conducting research. On the other hand, this information leakage may also seriously compromise the privacy of the entries in the databases. This can easily be seen on the (somewhat artificial) example of a more precise query of the form *'How many people in Germany who are called Alice WithAReallyComplicatedName and have insurance number* 123456789 *suffer from diabetes?'*. It is rather obvious that the result of such a query will be either 0 or 1, and that the correct answer immediately implies whether Alice WithAReallyComplicatedName has diabetes or not.

Differential privacy is a confidentiality property for database queries which allows for the release of statistical information about the content of a database without disclosing personal data. The variety of database queries and enforcement mechanisms has recently sparked the development of a number of mechanized proof techniques for differential privacy [36–39]. Personal data, however, are often spread across multiple databases and queries have to be jointly computed by multiple, possibly malicious, parties. Many cryptographic protocols have been proposed to protect the data in transit on the network and to achieve differential privacy in a *distributed*, adversarial setting. Proving differential privacy for such protocols is hard and, unfortunately, out of the scope of the aforementioned mechanized proof techniques.

1.3 Contributions

This thesis proposes three frameworks for the verification of security protocols and their implementations based on powerful types for security and privacy.

It introduces two novel affine type systems, namely AF7 and DF7, that allow for the automated verification of different security and privacy properties in cryptographic protocol implementations. We furthermore propose a generically applicable logical theory, which, based on pre- and post-conditions for security-critical code, guides existing type-checkers towards the verification of e-voting protocols.

1.3.1 AF7: A type system for resource-aware authorization policies

We propose AF7, the first type system that statically enforces the safety of cryptographic protocol implementations with respect to authorization policies expressed in *affine logic*. Such substructural logics can be used to express resource-aware properties that were out of the scope of previous type-based analysis techniques. AF7 builds on previous powerful type systems [31–34] that leverage general-purpose theorem proving techniques, extending them to support our fragment of intuitionistic affine logic. To protect affine formulas from duplication AF7 relies on the novel notion of "exponential serialization". We demonstrate the effectiveness of AF7 on two case studies. Furthermore, we propose a sound and complete algorithmic variant of the system called AF7$_{\text{alg}}$, which is the key to deriving an efficient implementation of our analysis technique.

1.3.2 A logical theory for the type-based analysis of electronic voting protocols

We present a novel approach for the automated analysis of e-voting protocols based on refinement type systems. Specifically, we design a generically applicable logical theory which, based on pre- and post-conditions for security-critical code, captures and guides the type-checker towards the verification of two fundamental properties of e-voting protocols, namely, vote privacy and verifiability. We provide a code-based cryptographic abstraction of the cryptographic primitives commonly used in e-voting protocols, showing how to make the underlying algebraic properties accessible to automated verification through logical refinements. We demonstrate the effectiveness of our approach by developing the first automated analysis of Helios, a popular web-based e-voting protocol, using an off-the-shelf type-checker.

1.3.3 DF7: A type system for distributed differential privacy

We propose a symbolic definition of differential privacy for distributed databases, which takes into account Dolev-Yao intruders and can be used to reason about compromised parties. We then introduce DF7, an affine, distance-aware type system to statically and automatically enforce this notion of distributed differential privacy in cryptographic protocol implementations. Our system builds on and extends a previous type system for the non-distributed case [36]. We also provide a sound and complete algorithmic variant of our type system called DF7$_{\mathrm{alg}}$ and tested our analysis technique on a recently proposed protocol for privacy-preserving web analytics: we discovered a new attack acknowledged by the authors, proposed a fix, and successfully type-checked the revised variant.

1.4 Outline

The dissertation is organized as follows: Chapter 2 presents the affine AF7 type system. Chapter 3 introduces our framework for the type-based verification of electronic voting protocols. Chapter 4 presents the affine DF7 type system. Chapter 5 concludes and Chapter 6 gives directions for future research.

The appendix consists of two parts: Appendix A contains the proofs of Chapter 2, while the proofs of Chapter 4 are given in Appendix B.

Part II

Type-Based Verification of Authorization Policies

2

AF7: A Type System for Resource-Aware Authorization Policies

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. Although successful in many respects, these type systems fall short of capturing resource-conscious properties that are crucial in large classes of modern distributed applications. In this chapter, we propose AF7, the first type system that statically enforces the safety of cryptographic protocol implementations with respect to authorization policies expressed in affine logic. Our type system draws on a novel notion of "exponential serialization" of affine formulas, a general technique to protect affine formulas from the effect of duplication. This technique allows to formulate an expressive logical encoding of the authentication mechanisms underpinning distributed resource-aware authorization policies. We discuss the effectiveness of our approach on two case studies: the EPMO e-commerce protocol and the Kerberos authentication protocol. We finally devise a sound and complete type-checking algorithm, which is the key to achieving an efficient implementation of our analysis technique.

Publication. In this chapter we present the work that was published under the title 'Affine Refinement Types for Secure Distributed Programming' in the ACM Transactions on Programming Languages and Systems [4] in 2015. An earlier version of this work was presented at the 2nd Conference on Principles of Security and Trust [3] in 2013, where it has been awarded the *EATCS Award* for the best theory paper at ETAPS. Preliminary results were presented at the 24th IEEE Computer Security Foundations Symposium [1] in 2011 and at the 7th International Symposium on Trustworthy Global Computing [2] in 2012. This line of research was a joint project with Michele Bugliesi, Stefano Calzavara, and Matteo Maffei. Both the author of this thesis and Stefano Calzavara contributed equally to the development of the AF7 type system and the corresponding soundness proofs as the main authors. These joint contributions and additionally his work on the proofs of exponential serialization also contributed to Stefano Calzavara's PhD thesis [40]. The details and proofs of the algorithmic variant AF7$_{\mathrm{alg}}$ are due to the author of this thesis and were not part of [40].

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 [31–33, 35]. 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 true.

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 [41]. A number of proposals have thus 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 [42, 43]. 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 [1].

Contributions In this chapter, we present AF7, an *affine refinement type system* for RCF [31], 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 framework.

We show the effectiveness of our approach on two case studies, namely the *EPMO* e-commerce protocol [44] and the *Kerberos* authentication protocol [45]. For both case studies we discuss the advantages in expressiveness enabled by the adoption of an underlying substructural logic.

Outline. Section 2.2 overviews the challenges and the most important aspects of our theory on a simple example. Section 2.3 reviews intuitionistic affine logic. Section 2.4 presents the meta-theory of exponential serialization. Section 2.5 reviews RCF and defines our notion of safety. Section 2.6 outlines the type system. Section 2.7 discusses encodings of network communication and our treatment of formal cryptography. Sections 2.8-2.9 present the case studies. Section 2.10 discusses the algorithmic formulation of our type system. Section 2.11 overviews the related work. Section 2.12 concludes.

For the proof of soundness of exponential serialization, which is due to Stefano Calzavara and Michele Bugliesi, we refer to [4, 40]. The proofs of the remaining main theorems are provided in the appendices: Section A.1 details a soundness proof for the AF7 type system and Section A.2 provides proofs of the soundness and completeness of the algorithmic version AF7_{alg} of the type system.

2.2 Overview of the framework

Our protocol specification language is an affine variant of RCF, a concurrent λ-calculus with message passing and refinement types originally introduced in [31] that we will refer to as $\mathrm{RCF}_{\mathrm{AF7}}$. We anticipate that $\mathrm{RCF}_{\mathrm{AF7}}$ is very expressive and can be mapped to a large subset of F#. For better readability, in the examples we use F#-like syntax with polymorphic types: our theoretical framework lacks full-fledged polymorphism, but that can be recovered by duplicating definitions at multiple monomorphic types when needed.

2.2.1 Protocol verification with (affine) refinement types

Verifying distributed authorization protocols with refinement types presupposes that protocols be annotated with security *assumptions* and *assertions*. The former are formulas that are assumed to hold at a given point in time, and they are employed to specify authorization policies and to encode the credentials available to request authorization. In contrast, assertions act as guards defining the properties to be entailed by the assumptions and the underlying policy, to grant authorization [31, 46, 47].

An example will help in making the discussion concrete. We introduce a system to place and ship orders in a distributed online service governed by a simple authorization policy, establishing that an order can be cleared for shipping to a user only if that user has indeed placed the order. For example, we could start by assuming the authorization policy encoded by the first-order formula: $\mathcal{P} \triangleq \forall x, y.(\mathsf{Order}(x, y) \Rightarrow \mathsf{Ship}(x, y))$. The security-annotated code corresponding to the online service scenario is given below:

```
let place_order = fun ch id item skey →
  assume Order(id,item);
  let pkt = sign skey (id,item) in send ch pkt

let ship_order = fun ch vkey →
  let pkt = recv ch in
    let (xc, xit) = verify vkey pkt in
      assert Ship(xc,xit)
```

The assumption Order*(id,item)* makes the required credential available to the *place_order* function, enabling the subsequent code to sign a request with the key *skey* and send it off over channel *ch*. Upon receiving the message, *ship_order* verifies the signature using the verification key *vkey*, retrieves the two components *xc* and *xit* of the request and asserts the formula Ship*(xc,xit)*.

A client and a server will execute the two functions, communicating on a shared channel *ch* and using a pair of corresponding signing and verification keys, as shown below (the server runs *ship_order* recursively to serve multiple requests):

```
let prot_spec ch =
  assume 𝒫;
  let sk = mksigkey () in
    let vk = mkverkey sk in
      let client = (place_order ch "alice" "book" sk) in
        let rec server = (ship_order ch vk) ↱ (server ch vk) in
```

$\mathit{client} \Rsh \mathit{server}$

The protocol specification given above may be proved *robustly* safe by existing refinement type systems: this ensures that the conjunction of all the assertions which will become active at run-time (i.e., $\mathsf{Ship}(\text{"}\mathit{alice}\text{"}, \text{"}\mathit{book}\text{"})$), is entailed by the active assumptions (i.e., $\mathcal{P}$, $\mathsf{Order}(\text{"}\mathit{alice}\text{"}, \text{"}\mathit{book}\text{"})$), despite the best efforts of an arbitrary opponent. Unfortunately, a closer look reveals that the authorization policy $\mathcal{P}$ is too weak to enforce desirable *resource-aware* access constraints: for instance, in our example the online service is presumably interested in ensuring that each user's order can be cleared and shipped only *once*, but in first-order logic we can prove:

$$\forall x, y.(\mathsf{Order}(x, y) \Rightarrow \mathsf{Ship}(x, y)), \mathsf{Order}(id, \mathit{item}) \vdash \mathsf{Ship}(id, \mathit{item}) \wedge \mathsf{Ship}(id, \mathit{item}),$$

i.e., a single payment by the user can lead to the same order being shipped twice, without violating the previous authorization policy and (robust) safety.

Remarkably, the desired resource-aware authorization policy can be naturally encoded in affine logic by assuming the formula: $\mathcal{P}_{okay} \triangleq\ !\forall x, y.(\mathsf{Order}(x, y) \multimap \mathsf{Ship}(x, y))$, where the bang modality (!) allows using the authorization policy arbitrarily many times in a proof, while the multiplicative implication ($\multimap$) ensures that formulas of the form $\mathsf{Order}(id, \mathit{item})$ are *consumed* when proving $\mathsf{Ship}(id, \mathit{item})$. Verifying the desired *injective* correspondence between placed and shipped orders amounts then just to reinterpreting the standard notion of (robust) safety by taking into account the *multiplicative* conjunction ($\otimes$) of the top-level assertions rather than the standard conjunction of first-order logic: roughly, this ensures that the (multi-)set of assumptions can be partitioned in different (multi-)sets, each proving one specific assertion, hence the same assumption is never used in the proof of two different assertions.

Extending refinement type systems to show compliance with respect to affine logic policies like $\mathcal{P}_{okay}$ is challenging. Technically, these type systems support a form of compositional reasoning enabled by the structure of the cryptographic key types, and the typing discipline enforced on them. Briefly, cryptographic key types are associated with refinement types of the form $\mathsf{Key}(\{x : T \mid F\})$, enforcing the following invariants: (i) to package a value $M : T$ with a key of this type, one must be able to prove $F(M)$ and consequently, (ii) upon extracting a value $w : T$ packaged under a key of this type, one may in turn assume the formula $F(w)$ to hold. These two invariants are enough to derive static proofs of robust safety in traditional refinement type systems drawing on classical and intuitionistic logics, but they fall short of providing the necessary guarantees in resource-conscious settings such as the one we consider here.

2.2.2 Exponential serialization for protecting affine formulas

Given the nature of affine formulas as consumable resources, an affine refinement type system must additionally provide protection against an unconstrained assumption of the refinement formulas conveyed by the key types [1]. For instance, when receiving a packet signed with a key of type $\mathsf{SigKey}(x : T, \{x : U \mid \mathsf{Order}(x, y)\})$, we must ensure that *each time* we verify the signature (and assume $\mathsf{Order}(x, y)$) at the receiver side, a corresponding assumption has indeed been introduced at the sender side.

Ensuring this kind of injective correspondence in distributed settings is known to require some protective measures, as an adversary may easily break it by mounting a replay attack and fool a receiver into deriving multiple assertions corresponding to one single assumption. We can see that in our running example: given the protocol specification defined above, assume we let it run over an untrusted network by passing the function *prot_ spec* as a parameter to the function *adversary* defined below, which intercepts the message by the client and sends it twice to the server:

```
let adversary prot =
  let ch = mkchan () in
    prot ch;
    let m = recv ch in (send ch m) ↾ (send ch m)
```

The replay attack mounted by the adversary breaks the desired injective correspondence between assumptions and assertions, since the system admits a run in which the adversary intercepts the message exchanged on *ch* and duplicates it, leading to two assertions $\mathsf{Ship}(\textit{"alice"}, \textit{"book"})$ being made against just one assumption $\mathsf{Order}(\textit{"alice"}, \textit{"book"})$. More technically, in affine logic we have:

$$!\forall x, y.(\mathsf{Order}(x, y) \multimap \mathsf{Ship}(x, y)), \mathsf{Order}(id, item) \nvdash \mathsf{Ship}(id, item) \otimes \mathsf{Ship}(id, item),$$

hence the protocol above is *not* robustly safe in our affine setting.

The problem we just outlined is, in fact, rather general and may be stated as follows: data exchanged over the network is inherently exposed to replays, hence their credentials, occurring as refinements of cryptographic key types, must be protected so that replicating the data does not duplicate the credentials. In the type system, this may be achieved by guarding the refinements of the key types with *control* formulas, which are guaranteed to be assumed in at most one point of the protocol code.

The resulting typing discipline leverages the underlying computational measures to counter replay attacks. Though the details vary for the different computational mechanisms, the intuition applies uniformly. The types of cryptographic keys are built around *guarded* refinements of the form:

$$\{\tilde{w} : \tilde{T}, \tilde{x} : \tilde{U} \mid !(C(\tilde{w}) \multimap F(\tilde{x}))\},$$

protecting the credential $F(\tilde{x})$ with the control formula $C(\tilde{w})$. In a nonce-handshake protocol, for instance, $\tilde{w}$ may represent a challenger-generated nonce, call it n, and $C(n)$ may be the corresponding guard assumed by the challenger, modeling that the nonce has been freshly generated. Upon receiving the nonce, a responder willing to transmit M will package the pair (n, M) under a key with the above type as a payload: intuitively, the receiver can then open the cryptographic packet to assume the implication above and derive the desired formula $F(M)$ by *consuming* the formula $C(n)$, which was never sent on the network and remained thus under the control of the challenger.

Notice that guarded refinement types as the one above contain an *exponential* formula prefixed by the bang modality, hence opening messages packaged under a key with this type more than once does not really provide additional information to the receiver and is perfectly safe. We call this packaging technique *exponential serialization*, as it provides us with a safe way to transmit payload with affine refinement types over an untrusted network, using an encoding based on exponential formulas.

2.2.3 Serializers for security type-checking

There is one problem left with the intuition above. A responder possessing the credential $F(M)$ and willing to prove it to the challenger will not be able to do so, as in affine logic an assumption $F(M)$ does not entail the guarded exponential formula $!(C(n) \multimap F(M))$, which the responder would need to prove to type-check the response. To close this gap, each affine assumption in the code must be associated with a corresponding *serializer*, to enable its use in the guarded refinements of the key types. Serializers have the general form:

$$!\forall \tilde{x}.\tilde{w}(F(\tilde{x}) \multimap !(C(\tilde{w}) \multimap F(\tilde{x}))),$$

and explicitly enable the transformation of the credential $F(\tilde{M})$ into its serialized form $!(C(\tilde{n} \multimap F(\tilde{M})))$, for appropriate terms $\tilde{n}$ and $\tilde{M}$.

Back to our example, assume we extend the protocol to include the nonce-handshake mentioned above:

```
let place_order' = fun ch1 ch2 id item skey →
  let nonce = recv ch1 in
    assume Order(id, item);
    let pkt = sign skey (id,item,nonce) in send ch2 pkt

let ship_order' = fun ch1 ch2 vkey →
  let mknonce = (fun () → let x = mkfresh () in assume N(x); x) in
    let nonce = mknonce () in
      send ch1 nonce;
      let pkt = recv ch2 in
        let (xc, xit ,xn) = verify vkey pkt in
          if (xn = nonce) then
            assert Ship(xc, xit)
          else
            failwith "unauthorized"
```

We assume to be given access to a library function *mkfresh* : unit → bytes, which generates fresh bit-strings. The function *mknonce*: unit → $\{x : \mathsf{bytes} \mid \mathsf{N}(x)\}$ is a wrapper around *mkfresh*, which additionally assumes the control formula $\mathsf{N}(x)$ over the returned value x. The new assumption is reflected by the *refined* return type of *mknonce*. Then, the typing of the signing and verification keys may be structured as follows:

$$\begin{aligned} skey &: \mathsf{SigKey}(\{x : \mathsf{string}, y : \mathsf{string}, z : \mathsf{bytes} \,|\, !(\mathsf{N}(z) \multimap \mathsf{Order}(x,y))\}) \\ vkey &: \mathsf{VerKey}(\{x : \mathsf{string}, y : \mathsf{string}, z : \mathsf{bytes} \,|\, !(\mathsf{N}(z) \multimap \mathsf{Order}(x,y))\}) \end{aligned}$$

conveying the affine formula $\mathsf{Order}(xc, xit)$ conditionally to the *guard* $\mathsf{N}(nonce)$ assumed by *ship_order'*. If the guard can be proved only once, $\mathsf{Order}(xc, xit)$ can also be retrieved only once, irrespectively of the number of signature verifications performed. To type-check the protocol, we need to assume the expected serializer:

$$\mathcal{S} \triangleq\ !\forall x, y.z.(\mathsf{Order}(x,y) \multimap\ !(\mathsf{N}(z) \multimap \mathsf{Order}(x,y))).$$

Overall, we get the following revised protocol:

```
let prot_spec' ch1 ch2 =
  assume $\mathcal{P}_{okay}$;
  assume $\mathcal{S}$;
  let sk = mksigkey () in
    let vk = mkverkey sk in
      let client ' = (place_order' ch1 ch2 "alice" "book" sk) in
        let rec server ' = (ship_order' ch1 ch2 vk) ↱ (server' ch vk) in
          client ' ↱ server'
```

We briefly discuss how the two protocol components type-check. We start from the server. Upon creating *nonce*, *server'* assumes the control formula $\mathsf{N}(nonce)$ based on the return type of the function *mknonce* by calling *ship_order'*. Upon verifying the received signature, it extracts the refinement $!(\mathsf{N}(xn) \multimap \mathsf{Order}(xc, xit))$ based on the type of the verification key. Then, from the assumption $\mathsf{N}(nonce)$ and the nonce-checking test $xn = nonce$ that protects the assertion, it derives $\mathsf{N}(xn)$. Now, with two $\multimap$ elimination steps, using the refinement above and the policy $\mathcal{P}_{okay}$, it derives the asserted formula $\mathsf{Ship}(xc, xit)$. As to the client, upon receiving the challenge, by calling the function *place_order'*, *client'* assumes the formula $\mathsf{Order}($*"alice"*, *"book"*$)$ and then signs the triple *(cid,item,nonce)* with *vk*. Typing the signature requires the serializer, which provides a direct way to prove the desired formula.

We notice here that serializers may be generated automatically for any given affine formula, and we prove that introducing them as additional assumptions is sound, in that it does not affect the set of entailed assertions, under the sufficient conditions discussed 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. We discuss these patterns in our case studies in Sections 2.8-2.9.

2.3 Review: affine logic

In our framework we focus on a simple, yet expressive, fragment of intuitionistic affine logic [43]. We presuppose an underlying signature Σ of predicate symbols, ranged over by p, and function symbols, ranged over by f. The syntax of terms t and formulas F is defined by the following productions:

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

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. The logical truth is written $\mathbf{1}$ and encoded as $() = ()$, where $()$ is the nullary function symbol encoding the RCF$_{\text{AF7}}$ "unit" value[1]. The negation of F, written $F^{\perp}$, is encoded as $F \multimap \mathbf{0}$, while inequality, written $t \neq t'$, is encoded as $(t = t')^{\perp}$. For simplicity, we do not consider disjunction and existential quantification: the logic considered here suffices for our purposes and we leave further extensions as future work.

$$\begin{array}{cccc}
\begin{array}{l}(\textsc{Ident})\\ F \vdash F\end{array} &
\begin{array}{l}(\textsc{Weak})\\ \dfrac{\Delta \vdash F'}{\Delta, F \vdash F'}\end{array} &
\begin{array}{l}(\textsc{Contr})\\ \dfrac{\Delta, !F, !F \vdash F'}{\Delta, !F \vdash F'}\end{array} &
\begin{array}{l}(\otimes\text{-}\textsc{Left})\\ \dfrac{\Delta, F_1, F_2 \vdash F'}{\Delta, F_1 \otimes F_2 \vdash F'}\end{array}
\end{array}$$

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

$$\begin{array}{cccc}
\begin{array}{l}(\forall\text{-}\textsc{Left})\\ \dfrac{\Delta, F\{t/x\} \vdash F'}{\Delta, \forall x.F \vdash F'}\end{array} &
\begin{array}{l}(\forall\text{-}\textsc{Right})\\ \dfrac{\Delta \vdash F \qquad x \notin fv(\Delta)}{\Delta \vdash \forall x.F}\end{array} &
\begin{array}{l}(!\text{-}\textsc{Left})\\ \dfrac{\Delta, F \vdash F'}{\Delta, !F \vdash F'}\end{array} &
\begin{array}{l}(!\text{-}\textsc{Right})\\ \dfrac{!\Delta \vdash F}{!\Delta \vdash !F}\end{array}
\end{array}$$

$$\begin{array}{ccc}
\begin{array}{l}(\textsc{False})\\ \mathbf{0} \vdash F\end{array} &
\begin{array}{l}(=\text{-}\textsc{Subst})\\ \dfrac{\exists \sigma = mgu(t,t') \Rightarrow \Delta\sigma \vdash F\sigma}{\Delta, t = t' \vdash F}\end{array} &
\begin{array}{l}(=\text{-}\textsc{Refl})\\ \Delta \vdash t = t\end{array}
\end{array}$$

Table 2.1: The entailment relation $\Delta \vdash F$ (AF7)

The entailment relation $\Delta \vdash F$ from multiset of formulas to formulas is given in Table 2.1. Observe that, in affine logic, rule (Weak) can be liberally applied to disregard formulas along a proof derivation, while rule (Contr) is restricted

[1] We mention here that RCF$_{\text{AF7}}$ terms can be encoded into the logic using the locally nameless representation of syntax with binders [48], as shown in [31].

to exponential formulas, allowing for 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 [42].

As informally discussed before, affine logic provides multiplicative counterparts of standard logical connectives: for instance, rule ($\otimes$-Right) states that 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 affine hypothesis in Δ is used either to prove F_1 or to prove F_2. Analogously, rule ($\multimap$-Left) formalizes the intuition that the 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.

Rule (!-Left) is often called the *dereliction* rule and allows exponential assumptions to be degraded to affine assumptions, which can be used at most once. Rule (!-Right), instead, is typically referred to as the *promotion* rule, which allows one to prove exponential formulas starting from the proof of an affine formula: the notation $!\Delta$ means that every formula in Δ must be of the form $!F$. The two rules for equality (=-Subst) and (=-Refl) are borrowed from [49]; in rule (=-Subst), if the terms t and t' are not unifiable, then we consider the premise as trivially fulfilled.

2.4 Metatheory of exponential serialization

Recall from Section 2.2.3 that we had to explicitly assume a serializer $\mathcal{S}$ to make our example protocol type-check. In principle, the introduction of this serializer among the assumed hypotheses could alter the intended semantics of the authorization policy $\mathcal{P}_{okay}$, due to the subtle interplay of formulas through the entailment relation defined in Table 2.1. Here, we isolate sufficient conditions under which exponential serialization leads to a sound protection mechanism for affine formulas. This contribution is due to Stefano Calzavara and Michele Bugliesi and we solely include it for the sake of completeness. More details and proofs can be found in [4, 40].

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 $p(t_1, \dots, t_n)$ 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:

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

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 logical consistency: these elements are used in our typed analysis, but we stipulate that they are never directly assumed in the protocol code (and thus never serialized).

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 2.1 (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:*

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

Definition 2.2 (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 $C_1 \multimap C_2, C_2 \multimap C_3$, where C_1, C_2, C_3 are built over distinct predicate symbols, 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 these circular dependencies among control formulas and simplify the proof of our soundness result, Theorem 2.1 below. To prove that result, we need a further definition:

Definition 2.3 (Controlled Multiset). *Let $\Delta = P_1, \ldots, P_m, S_1, \ldots, S_n$ be a stratified multiset of formulas. We say that Δ is* controlled *if and only if $\Delta \vdash C^k$ implies $k \leq 1$ for any control formula C.*

The intuition underlying the definition 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 for 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 condition above ensures.

Theorem 2.1 (Soundness of Serialization)**.** *Let $\Delta = P_1, \ldots, P_m$. If $\Delta' = \Delta, S_1, \ldots, S_n$ is controlled and $\Delta' \vdash P$, then $\Delta \vdash P$ for all payload formulas P.*

Proof. See [4, 40]. □

Notice that checking if a multiset of formulas is controlled may be difficult, since this depends on logical entailment, hence it may be not obvious when the theorem above can be applied. Fortunately, however, we can isolate a sufficient criterion to decide whether a multiset of formulas is controlled, based on a simple syntactic check.

Proposition 2.1 (Checking Control)**.** *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 controlled.*

Proof. See [4, 40]. □

2.5 Review of RCF$_{\mathbf{AF7}}$ and safety

We now review RCF$_{\text{AF7}}$ [31], a concurrent λ-calculus with message passing primitives, which provides the core language around which our theory is developed. We also formally introduce the resource-aware variant of the standard notion of safety for RCF, which we have been mentioning.

2.5.1 Syntax of RCF$_{\mathbf{AF7}}$

We assume collections of names (a, b, c, m, n) and variables (x, y, z). The syntax of values and expressions of RCF$_{\text{AF7}}$ is introduced in Table 2.2. The notions of free names and free variables arise as expected, according to the scope defined in the table.

Values include variables, unit, pairs, functions and constructions; constructors account for the creation of standard tagged unions and iso-recursive types. We also encode the boolean values $\mathsf{true} \triangleq \mathsf{inl}()$ and $\mathsf{false} \triangleq \mathsf{inr}()$. Expressions of RCF$_{\text{AF7}}$ 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.

2.5.2 Sematics of RCF$_{\mathbf{AF7}}$

The semantics is mostly standard. The function application $(\lambda x. E)\ N$ evaluates to $E\{N/x\}$; the syntactic equality check $M = N$ evaluates to true when M is equal to N and to false otherwise; the let expression $\mathsf{let}\ x = E\ \mathsf{in}\ E'$

$M, N ::=$		*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)
	$E \Rsh E'$	fork
	$a!M$	message send
	$a?$	message receive
	$\mathsf{assume}\ F$	assumption
	$\mathsf{assert}\ F$	assertion

Table 2.2: Syntax of RCF$_{\text{AF7}}$ expressions

first evaluates E to a value N and then behaves as $E'\{N/x\}$; the pair splitting $\mathsf{let}\ (x, y) = (M, N)\ \mathsf{in}\ E$ evaluates to $E\{M/x\}\{N/y\}$; and the pattern matching $\mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ E\ \mathsf{else}\ E'$ evaluates to $E\{N/x\}$ when M is equal to $h\ N$ for some N, while it evaluates to E' otherwise. We then have some constructs reminiscent of process algebras: 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*" that are used in the code of our examples and that we further explain in Section 2.7.1. Assumptions and assertions are stuck expressions, which are just needed to state our safety notion (see below). The formal semantics of RCF$_{\text{AF7}}$ expressions is defined by the reduction rules in Table 2.3.

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.

$(\lambda x.\, E)\ N \to E\{N/x\}$	(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)
$a!M \upharpoonright a? \to M$	(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)
$E \upharpoonright E'' \to E' \upharpoonright E'' \quad \text{if } E \to E'$	(Red Fork 1)
$E'' \upharpoonright E \to E'' \upharpoonright E' \quad \text{if } E \to E'$	(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 $\mathrm{RCF}_{\mathrm{AF7}}$

2.5.3 Resource-aware safety

We are now ready to adapt the formal notion of safety defined for $\mathrm{RCF}_{\mathrm{AF7}}$ expressions to our resource-aware setting. Intuitively, an expression E is safe if, 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 and it is convenient to introduce the notion of *structure* for this purpose.

Let e denote an *elementary* expression, i.e., 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 multiset of assumed formulas F_i; (2) a multiset of asserted formulas F'_j; (3) a series of messages M_k sent on channels but not yet received; and (4) a series of elementary expressions e_ℓ being evaluated in parallel contexts. The definition of a structure $\mathbf{S}$ is given in Table 2.5. Structures are convenient, since their syntactic form already exhibits all the necessary ingredients to state a simple notion of *static safety*, the basic building block for safety.

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

Lemma 2.1 (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 2.4 (Safety)**.** *A closed expression E is* safe *if and only if, for all E' and $\mathbf{S}$, if $E \to^* E'$ 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 $\mathrm{RCF}_{\mathrm{AF7}}$ which does not contain any assumption

$E \Rrightarrow E$	(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 $a \notin \mathit{fn}(E')$	(Heat Res Fork 1)
$(\nu a)E \Rsh E' \Rrightarrow (\nu a)(E \Rsh E')$ if $a \notin \mathit{fn}(E')$	(Heat Res Fork 2)
$\mathsf{let}\ x = (\nu a)E\ \mathsf{in}\ E' \Rrightarrow (\nu a)(\mathsf{let}\ x = E\ \mathsf{in}\ E')$ if $a \notin \mathit{fn}(E')$	(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 $\mathrm{RCF}_{\mathrm{AF7}}$

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

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

Table 2.5: Structures and static safety

or 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 in practice, since we want to verify application code with respect to the security annotations placed therein. We note that security annotations are simply considered a tool for verification but that they hold no semantic meaning and are thus not necessary for the opponent code.

Definition 2.5 (Robust Safety)**.** *A closed expression E is* robustly safe *if and only if, for any opponent O, the application $O\ E$ is safe*[2]*.*

2.6 The AF7 type system

Our refinement type system builds on previous work by Bengtson et al. [31], extending it to guarantee the correct usage of affine formulas and to enforce our revised notion of (robust) safety.

[2] Here, we use the standard syntactic sugar $O\ E$ for the expression $\mathsf{let}\ x = O\ \mathsf{in}\ \mathsf{let}\ y = E\ \mathsf{in}\ x\ y$.

2.6.1 Types, typing environments, and base judgements

The syntax of types is defined in Table 2.6.1. 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
	$x : T \to U$	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 (scope of α is T)
	α	type variable
	$\{x : T \mid F\}$	refinement type (scope of x is F)

Table 2.6: Syntax of types (AF7)

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 $x : T \to U$ 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.[3] Type $\mathsf{bool} \triangleq \mathsf{unit} + \mathsf{unit}$ is inhabited by $\mathsf{true} \triangleq \mathsf{inl}()$ and $\mathsf{false} \triangleq \mathsf{inr}()$.

The 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 as follows: $dom(\alpha) = \{\alpha\}$; $dom(\alpha :: k) = \{\alpha\}$; $dom(a \updownarrow T) = \{a\}$; $dom(x : T) = \{x\}$; and $dom(\mu_1, \ldots, \mu_n) = dom(\mu_1) \cup \ldots \cup dom(\mu_n)$. The set of *free* variables and free names is denoted by *fnfv*. The definition is standard.

We first discuss the base judgements of the type system. 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 will use the function ψ to place the structural type information in Γ and the function *forms* to place the associated refinements in Δ. We also write $\Gamma; \Delta \vdash T$ to denote that type T is well-formed in $\Gamma; \Delta$ and $\Gamma; \Delta \vdash F$ when the formulas

[3] Note that other types built over Un are available to the opponent through subtyping.

in Δ entail the formula F. We often abuse notation and write $\Gamma;\Delta \vdash F_1,\ldots,F_n$ to stand for $\Gamma;\Delta \vdash F_1 \otimes \ldots \otimes F_n$, with the proviso that $\Gamma;\Delta \vdash \emptyset$ is equivalent to $\Gamma;\Delta \vdash \mathbf{1}$. A complete formal definition of the described elements is given in Table 2.6.1 below.

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

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

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

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

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

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

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

Table 2.7: Auxiliary functions and base judgements (AF7)

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 with affine types[4]. 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. However, a simple splitting of the formulas in Δ would lead to a very restrictive type system. To illustrate, let $\Delta \triangleq A, A \multimap\, !B$: if we just distributed the formulas A and $A \multimap\, !B$ between two distinct subderivations, then the formula $!B$ would be available only in (at most) one subderivation, despite it being an exponential formula, which we may want to use arbitrarily often during type-checking.

[4] In Section 2.6.8 we thoroughly discuss why this does not involve any loss in expressiveness, by showing an encoding of affine types through exponential serialization.

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 $\Gamma;\Delta'$. This relation is defined by rule (Rewrite) 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 $\Delta \vdash \mathbf{1}$. Coming back to our previous example, notice that we have $A, A \multimap !B \vdash !B \otimes !B$ in affine logic, hence we can obtain two copies of $!B$ upon rewriting and distribute them between two distinct subderivations upon type-checking. As we will explain in Section 2.6.5, for soundness reasons we will often rely on rewriting of the form $\Gamma;\Delta \hookrightarrow \Gamma;!\Delta'$, where $!\Delta'$ is a so-called *exponential* environment, i.e., an environment of the form $!F_1,\ldots,!F_n$.

The adoption of the environment rewriting relation as an house-keeping device for the formulas in 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 substructural logic was first proposed by Mandelbaum et al. [51], even though their solution is technically different from ours. Namely, the authors of [51] 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 2.10.

2.6.3 Kinding

Security type systems often rely on a kinding relation to discriminate whether or not messages of a specific type may be sent to the attacker or generated by 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. We let $\overline{\mathsf{pub}} \triangleq \mathsf{tnt}$ and $\overline{\mathsf{tnt}} \triangleq \mathsf{pub}$.

The complete kinding relation is given in Table 2.6.3. Most of the rules resemble those presented in other security type systems [31–33] 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. Here, we just point out some simple observations, which should hopefully guide the reader in understanding a few important aspects.

(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)
$$\frac{\Gamma;!\Delta_1 \vdash T :: \overline{k} \qquad \Gamma, x : \psi(T);!\Delta_2 \vdash U :: k \qquad \Gamma;\Delta \hookrightarrow \Gamma;!\Delta_1,!\Delta_2}{\Gamma;\Delta \vdash x : T \to U :: k}$$

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

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

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

(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)
$$\frac{\Gamma;\Delta_1 \vdash \psi(T) :: \mathsf{tnt} \qquad \Gamma, y : \psi(T);\Delta_2 \vdash \mathit{forms}(y : T) \qquad \Gamma;\Delta \hookrightarrow \Gamma;\Delta_1,\Delta_2 \qquad T \text{ refined}}{\Gamma;\Delta \vdash T :: \mathsf{tnt}}$$

Table 2.8: Kinding relation (AF7)

The type unit is assumed to be both public and tainted by (Kind Unit). According to (Kind Pair), a pair type is public if both its components are public and can be disclosed to the opponent. Conversely, by the same rule, a pair type is tainted if both its components are tainted, since, if even a single component of the pair is untainted, then the pair cannot come from the opponent. The kinding of sum types (Kind Sum) behaves analogously. By rule (Kind Fun) a function type is public (thus available to the attacker) only if its return type is public (otherwise $\lambda x. M_{\text{secret}}$ could be public and leak a secret to the attacker) and its argument type is tainted such that it can be called by the attacker. The treatment of tainted function types is dual. To give kind k to an iso-recursive type with a bound variable α, the rule (Kind Rec) proceeds recursively and extends the typing environment in the premise with the kinding annotation $\alpha :: k$. These kinding annotations are used when kinding a type variable (Kind Var). By

(Kind Refine Public) a refinement type is public if the structural type it refines is public, while by (Kind Refine Tainted) it is tainted if its structural information is tainted and its refinements are entailed by the typing environment.

2.6.4 Subtyping

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 safely used in place of values of type U. The complete presentation of the subtyping relation can be found in Table 2.6.4.

$$\textsc{(Sub Refl)}\quad \dfrac{\Gamma;\Delta \vdash T}{\Gamma;\Delta \vdash T <: T} \qquad \textsc{(Sub Pub Tnt)}\quad \dfrac{\begin{array}{c}\Gamma;\Delta_1 \vdash T :: \mathsf{pub} \qquad \Gamma;\Delta_2 \vdash U :: \mathsf{tnt}\\ \Gamma;\Delta \hookrightarrow \Gamma;\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta \vdash T <: U}$$

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

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

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

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

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

Table 2.9: Subtyping relation (AF7)

We first note that subtyping is reflexive by (Sub Refl). Furthermore, 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 and sum types are covariant (cf. (SUB PAIR) and (SUB SUM)), while function types are contravariant in their arguments and covariant in their return types (cf. (SUB FUN)). 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 iso-recursive types (SUB POS REC) is borrowed from [32,33] 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 [32,33] for further discussion on this technical point.

The most interesting subtyping rule in Table 2.6.4 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 discarding 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 further qualified by the information encoded by the formula F. The second rule, instead, generalizes the substitution principle underlying subtyping to the refinement formulas: for instance, we have $\emptyset; \varepsilon \vdash \{x : \mathsf{Un} \mid x = 5\} <: \{x : \mathsf{Un} \mid x > 0\}$, since the logical 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 duplicated and we could prove, for instance: $\emptyset; \varepsilon \vdash \{x : \mathsf{Un} \mid F\} <: \{z : \{x : \mathsf{Un} \mid F\} \mid F\}$ by using (SUB REFL) in the left premise of (SUB REFINE WRONG). 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 an affine setting, previous work [1] highlighted that the technical treatment of these rules is rather complicated, and we find rule (SUB REFINE) more convenient for proofs. The previous discussion should have also provided an intuition on the reasons behind a slightly more restrictive treatment for subtyping pairs and functions with respect to the original RCF paper, i.e., we must take care in applying the refinement stripping function ψ before extending the typing environment in the second premise of the corresponding rules.

2.6.5 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 2.6.5.

$$\text{(Val Var)}\quad \frac{\Gamma; \Delta \vdash \diamond \qquad (x : T) \in \Gamma}{\Gamma; \Delta \vdash x : T} \qquad\qquad \text{(Val Unit)}\quad \frac{\Gamma; \Delta \vdash \diamond}{\Gamma; \Delta \vdash () : \mathsf{unit}}$$

$$\text{(Val Fun)}\quad \frac{\begin{array}{c}\Gamma, x : \psi(T); !\Delta', \mathit{forms}(x : T) \vdash E : U \\ \Gamma; \Delta \hookrightarrow \Gamma; !\Delta'\end{array}}{\Gamma; \Delta \vdash \lambda x. E : x : T \to U}$$

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

$$\text{(Val Refine)}\quad \frac{\begin{array}{c}\Gamma; \Delta_1 \vdash M : T \qquad \Gamma; \Delta_2 \vdash F\{M/x\} \\ \Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2\end{array}}{\Gamma; \Delta \vdash M : \{x : T \mid F\}} \qquad\qquad \text{(Val Inl)}\quad \frac{\begin{array}{c}\Gamma; !\Delta' \vdash M : T \qquad \Gamma; !\Delta' \vdash U \\ \Gamma; \Delta \hookrightarrow \Gamma; !\Delta'\end{array}}{\Gamma; \Delta \vdash \mathsf{inl}\ M : T + U}$$

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

Table 2.10: Typing rules for values (LTS)

The rules for variable and unit typing are standard: variables are typed by looking up their type binding in the typing environment Γ using (Val Var); the unit value can be given type unit under any well-formed environment using (Val Unit). 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 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 include 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 $!F_1, \ldots, !F_n$. Indeed, 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 duplication of F upon repeated pair splitting operations on (M, N). Similar restrictions apply also to sum types (cf. (Val Inl) and (Val Inr)) and iso-recursive types (cf. (Val Fold)).

Notice that allowing for affine refinements, but forbidding affine types, confines the problem of resource management to the formula environment Δ, thus simplifying the technical development of the type system, as well as its algorithmic variant. In Section 2.6.8 we explain how our exponential serialization technique can be leveraged to encode affine types in our framework, hence our choice does not lead to any loss of expressiveness.

2.6.6 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 2.6.6.

Several typing rules make use of the *extraction* relation $E \rightsquigarrow [\Delta \mid D]$ that destructively collects all the assumed formulas Δ from the expression E and returns the expression D obtained by purging E of its assumptions. The relation is defined in Table 2.6.6 and will be explained further in the context of rule Exp Fork.

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. The rule for typing function applications (Exp Appl) divides the formula environment Δ among its premises and checks that the type of the argument corresponds to the expected function argument type; in the return type we substitute the argument to the variable bound in the function type, thus implementing a form of value dependent typing. 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 technique is used also in (Exp Match) and (Exp Eq). The treatment of channels is mostly standard: For each new channel a, a message type is determined ($a \updownarrow T$) and added to the typing environment Γ (cf. Exp Res) that is used to type-check the remaining expression. The rules for sending (Exp Send) and receiving (Exp Recv) messages on such channel assure that the sent/received messages have the correct type. Rule (Exp Assert) is standard and requires an asserted formula F to be derivable from the formulas collected by the typing environment: in fact, these formulas under-approximate the formulas which will be assumed at run-time. As we will see in the explanation of the rule (Exp Fork) below, due to the affine nature of the logic, the treatment of assumptions is a delicate task. Assumptions can be typed using either rule (Exp True) or (Exp Assume). The former describes the trivial case of a truth assumption $\mathbf{1}$ that is always given type unit, the latter is used for more complex formulas F, which are added to the formula environment Δ. Intuitively, the intended usage of these rules to type-check an assumption $\mathsf{assume}\ F$ with type T is as follows: (1) Prove $\Gamma; \Delta, F \vdash \mathsf{assume}\ \mathbf{1} : \mathsf{unit}$ by rule (Exp True); (2) Refine the type unit into T by subtyping; (3) Use (Exp Assume) to conclude $\Gamma; \Delta \vdash \mathsf{assume}\ F : T$.

The most complex rule is (Exp Fork): intuitively, when type-checking the

(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)

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

(Exp Split)

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

(Exp Match)

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

(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 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 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 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 2.11: Typing rules for expressions (AF7)

parallel expressions $E_1 \Rsh E_2$, assumptions in E_1 can be safely 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 by 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. 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: indeed, notice that the latter is not safe according to Definition 3.1.

The extraction relation $E \leadsto^{\widetilde{a}} [\Delta \mid D]$ is formally defined in Table 2.6.6. Note that we annotate the arrow with a list of names $\widetilde{a}$ to prevent formulas containing free names from being extracted outside the scope of the respective binders. For instance, in the expression $((\nu a)\mathsf{assume}\ F(a)) \Rsh \mathsf{assert}\ F(a)$ we do not want to use the assumption to type-check the parallel assertion, since the scope of the name a is limited to the assumption itself. The extraction relation is used to type-check any expression possibly containing "active" assumptions, i.e., lets (cf. (Exp Let)), restrictions (cf. (Exp Res)), and assumptions themselves (cf. (Exp Assume), which hardcodes the extraction).

(Extr Fork)

$$\frac{E_1 \leadsto^{\widetilde{a}} [\Delta_1 \mid D_1] \qquad E_2 \leadsto^{\widetilde{a}} [\Delta_2 \mid D_2]}{E_1 \Rsh E_2 \leadsto^{\widetilde{a}} [\Delta_1, \Delta_2 \mid D_1 \Rsh D_2]}$$

(Extr Let)

$$\frac{E_1 \leadsto^{\widetilde{a}} [\Delta \mid D_1]}{\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \leadsto^{\widetilde{a}} [\Delta \mid \mathsf{let}\ x = D_1\ \mathsf{in}\ E_2]}$$

(Extr Res)

$$\frac{E \leadsto^{a,\widetilde{b}} [\Delta \mid D]}{(\nu a)E \leadsto^{\widetilde{b}} [\Delta \mid (\nu a)D]}$$

(Extr Assume)

$$\frac{F \neq \mathbf{1} \qquad fn(F) \cap \{\widetilde{a}\} = \emptyset}{\mathsf{assume}\ F \leadsto^{\widetilde{a}} [F \mid \mathsf{assume}\ \mathbf{1}]}$$

(Extr Exp)

$$\frac{\text{no other rule applies}}{E \leadsto^{\widetilde{a}} [\emptyset \mid E]}$$

Table 2.12: The extraction relation (AF7)

2.6.7 Formal results

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

Theorem 2.2 (Safety). *If $\varepsilon; \emptyset \vdash E : T$, then E is safe.*

Proof. See Appendix A.1. □

Theorem 2.3 (Robust Safety). *If $\varepsilon; \emptyset \vdash E : \mathsf{Un}$, then E is robustly safe.*

Proof. See Appendix A.1. □

Theorem 3.1 above and Theorem 2.1 (Soundness of 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 their serializers $S_1, \dots, S_n$. Let then $E^\star = \mathsf{assume}\ S_1 \otimes \cdots \otimes S_n \Rsh E$ be the original expression extended with the serializers. By Theorem 3.1, if $\varepsilon; \emptyset \vdash E^\star : \mathsf{Un}$, then $E^\star$ is robustly safe. By Theorem 2.1, 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 controlled.

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 sufficient condition provided by Proposition 2.1. Actually, we observe that the syntactic criterion proposed by the proposition becomes a *semantic* property of the program to type-check, since programs contain variables to be replaced at run-time: we will discuss for our examples how we verify that the typing environment satisfies the conditions required for robust safety.

2.6.8 Discussion: encoding affine types

We now discuss how we can take advantage of exponential serialization to encode affine types in our type system. For the sake of simplicity, we focus on the encoding of affine pairs, but the same ideas applies uniformly to other data types (i.e., tagged unions and iso-recursive types).

Consider the typing environment $\Gamma; \Delta \triangleq x : \mathsf{Un}, y : \mathsf{Un}; A(x), B(y)$. Standard refinement type systems [31] 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 of the continuation. 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 $A'(x) \triangleq !(C_1(x) \multimap A(x))$ and $B'(y) \triangleq !(C_2(y) \multimap B(y))$ 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. In fact, notice that we have:

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

hence we can prove the following type judgement:

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

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 $C_1(x)$ and $C_2(y)$ 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 (see [52] for a thorough discussion on this point).

2.7 A library for communication and cryptography in RCF$_{\mathbf{AF7}}$

In this section we describe the primitives for communication that we use throughout the examples in this work and discuss how we encode cryptography using sealing. We note that our encoding of both communication and cryptography benefits from the notion of exponential serialization: we will never use channels, references, or cryptographic operations directly for messages with affine refinements, but we instead rely on exponentially serialized versions of such refinements. Formally, our libraries build on so-called *exponential types* that do not carry an affine refinement and are defined in Table 2.7. Since these types do not need to be protected from replication we can immediately leverage existing non-affine libraries [31].

For the sake of simplicity, the definitions of the necessary functions and types are parametric in a type variable γ used to denote exponential types. We recall, however, that our system does not support full polymorphism, but we can recover its effects by replicating library code to specialize it to the different types we need. Most of the content of this section is taken from [31] and included for the reader's convenience to make the chapter self-contained.

2.7.1 An encoding of channels and messaging

In RCF$_{\text{AF7}}$ channels are not values, hence they cannot be shared dynamically among principals. That same effect may however be recovered with the follow-

$$T \text{ exponential if } \begin{cases} T \in \{\mathsf{unit}, \alpha\} & \\ U \text{ exponential} & \text{for } T = \{x : U \mid !F\} \\ T_1 \text{ exponential and } T_2 \text{ exponential} & \text{for } T = x : T_1 \to T_2 \\ T_1 \text{ exponential and } T_2 \text{ exponential} & \text{for } T = x : T_1 * T_2 \\ T_1 \text{ exponential and } T_2 \text{ exponential} & \text{for } T = T_1 + T_2 \\ U \text{ exponential} & \text{for } T = \mu\alpha.\, U \end{cases}$$

Table 2.13: Exponential types (AF7)

ing encoding of channels for messages of exponential type γ (and the associated primitives for message passing).

We report both the communication interface and its implementation below.

```
type Ch(γ)  = (γ → unit) * (unit → γ)
val mkchan : unit → Ch(γ)
val send : Ch(γ) → γ → unit
val recv  : Ch(γ) → γ

let mkchan = fun _ → (new a)(fun x → a!x, fun _ → a?)
let send = fun c x → let (s, r) = c in s x
let recv  = fun c → let (s, r) = c in r ()
```

We note that references can be encoded analogously.

```
type Ref(γ)  = Ch(γ)
val mkref : γ → Ref(γ)
val setref  : Ref(γ) → γ → unit
val deref  : Ref(γ) → γ

let mkref = fun x → let r = mkchan () in send r x; r
let setref  = fun r x  → let _  =  recv r in send r x
let deref  = fun r  → let x =  recv r in send r x; x
```

In the following we typically write “*r := v*” for “*setref r v*” and “*!r*” for “*deref r*”.

Note that the code for dereferencing a reference will not type-check for types that are not exponential, since the value retrieved from the reference is used twice: it is stored back into the reference and returned. Without the serialization approach, we would thus need to change the implementation, for instance, by using destructive references that erase their content after a read.

2.7.2 A sealing-based encoding of cryptography

Formal cryptography can be encoded inside RCF$_{\mathrm{AF7}}$ in terms of *sealing* [53,54]. 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 such cryptographic schemes we identify the key with the seal, i.e., we give access to both the sealing and the unsealing functions to any owner of the key and we let $\mathsf{SymKey}(T) \triangleq (T \rightarrow \mathsf{Un}) * (\mathsf{Un} \rightarrow T)$. Different cryptographic primitives, like public key encryptions and signature schemes, can be encoded following the same recipe: for instance, since the owner of a signing key is typically able to verify her own signature, the sealing-based abstraction of a signing key may consist of both the sealing and the unsealing functions, and be given type $\mathsf{SigKey}(T) \triangleq (T \rightarrow \mathsf{Un}) * (\mathsf{Un} \rightarrow T)$. The corresponding verification key, instead, should comprise only the unsealing function and be given type $\mathsf{VerKey}(T) \triangleq \mathsf{Un} \rightarrow T$. The functions "*sign*" and "*verify*" introduced in Section 2.2 can then be straightforwardly implemented: *sign* M N just extracts the first component of M and calls it with parameter N, while *verify* M N simply invokes M with parameter N.

As stated above, another important benefit of exponential serialization is that we can immediately leverage the sealing-based cryptographic library proposed by Bengtson et al. [31], since we will define cryptographic operations to be performed only on messages of exponential type. 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, hence the dereferencing/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 [55]. Instead, with our approach, the unsealing function does not need to be changed: we can invoke it an arbitrary number of times to retrieve the payload, but the associated refinements will be retrieved at most once through exponential serialization.

We give full details of the cryptographic API used throughout this chapter (just the types, not the code) below.

type $\mathsf{Seal}(\gamma) \;\; = (\gamma \rightarrow \mathsf{Un}) * (\mathsf{Un} \rightarrow \gamma)$
type $\mathsf{SealRef}(\gamma) = \mathsf{Ref}(\mathsf{List}(\gamma * \mathsf{Un}))$

val *mkseal :* $\mathsf{string} \rightarrow \mathsf{Seal}(\gamma)$
val *seal :* $\mathsf{SealRef}(\gamma) \rightarrow \gamma \rightarrow \mathsf{Un}$
val *unseal :* $\mathsf{SealRef}(\gamma) \rightarrow \mathsf{Un} \rightarrow \gamma$

```
type SymKey(γ) = Sym of Seal(γ)
val mksymkey : unit → SymKey(γ)
val sencrypt : SymKey(γ) → γ → Un
val sdecrypt : SymKey(γ) → Un → γ

type SigKey(γ) = SK of Seal(γ)
type VerKey(γ) = VK of (Un → γ)
val mksigkey : unit → SigKey(γ)
val mkverkey : SigKey(γ) → VerKey(γ)
val sign  : SigKey(γ) → γ → Un
val verify  : VerKey(γ) → Un → γ

type DecKey(γ) = DK of Seal(γ)
type EncKey(γ) = EK of (γ → Un)
val mkdeckey : unit → DecKey(γ)
val mkenckey : DecKey(γ) → EncKey(γ)
val encrypt : EncKey(γ) → γ → Un
val decrypt : DecKey(γ) → Un → γ
```

2.8 Example: EPMO

We are finally ready to see our type system at work. We consider a variant of *EPMO*, a nonce-based e-payment protocol proposed by Guttman et al. [44].

2.8.1 Protocol description

The protocol narration is informally represented in Figure 2.1 (the meaning of the security annotations is explained below).

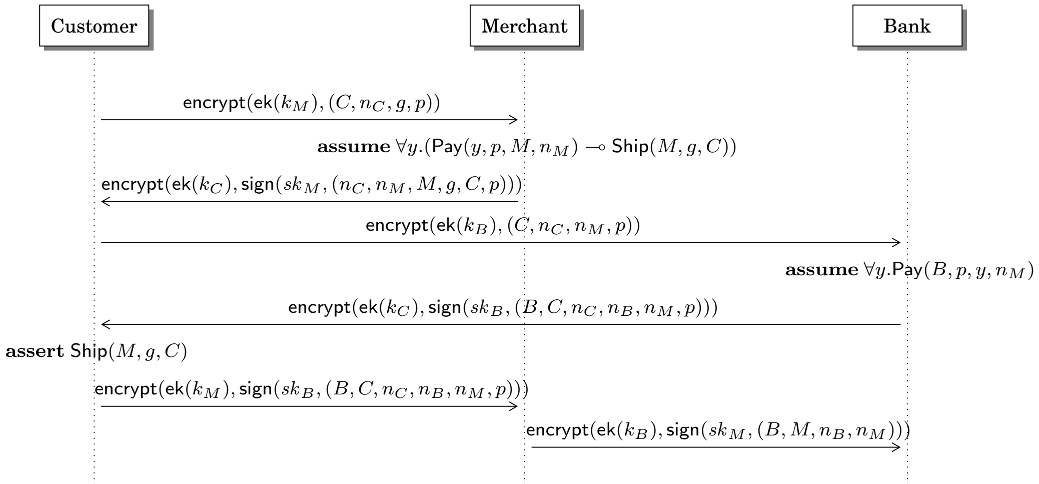

Customer
Merchant
Bank
$\mathsf{encrypt}(\mathsf{ek}(k_M), (C, n_C, g, p))$
assume $\forall y.(\mathsf{Pay}(y, p, M, n_M) \multimap \mathsf{Ship}(M, g, C))$
$\mathsf{encrypt}(\mathsf{ek}(k_C), \mathsf{sign}(sk_M, (n_C, n_M, M, g, C, p)))$
$\mathsf{encrypt}(\mathsf{ek}(k_B), (C, n_C, n_M, p))$
assume $\forall y.\mathsf{Pay}(B, p, y, n_M)$
$\mathsf{encrypt}(\mathsf{ek}(k_C), \mathsf{sign}(sk_B, (B, C, n_C, n_B, n_M, p)))$
assert $\mathsf{Ship}(M, g, C)$
$\mathsf{encrypt}(\mathsf{ek}(k_M), \mathsf{sign}(sk_B, (B, C, n_C, n_B, n_M, p)))$
$\mathsf{encrypt}(\mathsf{ek}(k_B), \mathsf{sign}(sk_M, (B, M, n_B, n_M)))$

Figure 2.1: 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, $\mathsf{ek}(k_M)$ (which we use as shorthand for "*mkenckey* k_M" throughout the example), and includes a fresh nonce, n_C. If M agrees to proceed in the transaction by providing a response signed with the signing key sk_M, 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.

2.8.2 Protocol analysis and challenges

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 ones.

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 will ship the goods g to C. After the first step of the protocol, we let the merchant M state the formula $\forall y.(\mathsf{Pay}(y, p, M, n_M) \multimap \mathsf{Ship}(M, g, C))$, to signify that she does not care about which bank is going to authorize the payment, but, as long as there is one authorizing bank, she will ship the good g to the client C at the end of the transaction. Conversely, after the appropriate checks on the client's account, we let the bank B assume the formula $\forall y.\mathsf{Pay}(B, p, y, n_M)$, to model that she authorizes the payment for the transaction n_M to any merchant chosen by the client. These two credentials allow the customer C to assert the formula $\mathsf{Ship}(M, g, C)$, a formal assurance on the validity of the transaction.

2.8.3 Type-checking the customer

The protocol code for the customer, enriched with the most relevant type annotations and the necessary serializers, is shown below. For the sake of readability, we use again F#-like syntax and some standard syntactic sugar like tuples, refined tuple types, algebraic types, and pattern matchings: all these can be encoded in $\mathrm{RCF}_{\mathrm{AF7}}$ and AF7 using standard techniques [31].

```
(* Serializer for M, needed to type-check M *)
assume !∀xp, xM, xnM, xg, xC, xnC.
```

```
            (∀y.(Pay(y, xp, xM, xnM) ⊸ Ship(xM, xg, xC)) ⊸
            !(N1(xnC) ⊸ (∀y.(Pay(y, xp, xM, xnM) ⊸ Ship(xM, xg, xC))))

(* Serializer for B, needed to type-check B *)
assume !∀yB, yp, ynC, ynM.
            (∀y.(Pay(yB, yp, y, ynM)) ⊸ !(N2(ynC) ⊸ (∀y.(Pay(yB, yp, y, ynM))))

(* Typing the message from M to C *)
type MsgMC = MsgMC of (xnC : bytes ∗ xnM : bytes ∗ xM : string ∗ xg : string
  ∗ xC : string ∗ xp : int)
  {!(N1(xnC) ⊸ ∀y.(Pay(y, xp, xM, xnM) ⊸ Ship(xM, xg, xC))}

(* Typing the message from B to C *)
type MsgBC = MsgBC of (yB : string ∗ yC : string ∗ ynC : bytes ∗ ynB : bytes
  ∗ ynM : bytes ∗ yp : int){!(N2(ynC) ⊸ ∀y.(Pay(yB, yp, y, ynM))}

(* Generate transaction identifiers *)
let mktid : unit → {x : bytes | N1(x) ⊗ N2(x)} = fun () →
  let xf = mkfresh () in assume (N1(xf) ⊗ N2(xf)); xf

(* Customer code *)
let cust C addC M addM B addB g p kC ekM ekB
          (vkM: VerKey(MsgMC + MsgMB) (vkB: VerKey(MsgBC)) =
  let nC = mktid () in
  (* N1(nC) and N2(nC) hold true *)
  let msgCM1 = encrypt ekM (C, nC, g, p) in send addM msgCM1;
  let signMC = decrypt kC (receive addC) in
  let plainMC = verify vkM signMC in
  match plainMC with MsgMC (=nC, xnM, =M, =g, =C, =p) →
      (* !(N1(nC) ⊸ ∀y.(Pay(y, p, M, xnM) ⊸ Ship(M, g, C)) holds true *)
      let msgCB = encrypt ekB (C, nC, xnM, p) in send addB msgCB;
      let signBC = decrypt kC (receive addC) in
      let plainBC = verify vkB signBC in
        match plainBC with MsgBC (=B, =C, =nC, xnB, =xnM, =p) →
          (* !(N2(nC) ⊸ ∀y.(Pay(B, p, y, xnM)) holds true *)
          assert Ship(M, g, C);
          let msgCM2 = encrypt ekM signBC 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 $\mathsf{N1}(nC)$ and $\mathsf{N2}(nC)$, later employed to authenticate the 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 this type, since it is the one needed to type-check the code of C: the refined formula in the corresponding type definition is retrieved upon verification of *signMC* and describes the promise by M to ship the goods as soon as the requested payment has been authorized by any bank chosen by the client.

We finally 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: this statement is available after verifying the message *signBC*. 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.

Notice that, to conclude that the code actually respects the authorization policy despite the introduction of the serializers, we also have to show that the program ensures the invariant that each control formula is assumed at most once, corresponding to the sufficient condition for control dictated by Proposition 2.1. This is an easy task to carry out, since we can just observe that control formulas are only assumed in the body of the *mktid* function, which in turn only performs these assumptions over the results of the *mkfresh* function for the generation of fresh bitstrings.

2.9 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 these identifiers are not just checked multiple times, but also by different parties. This is exactly the case for the mutual authentication step of the Kerberos protocol [45].

2.9.1 Protocol description

An informal narration of the protocol is shown in Figure 2.2 (the meaning of the security annotations is explained below).

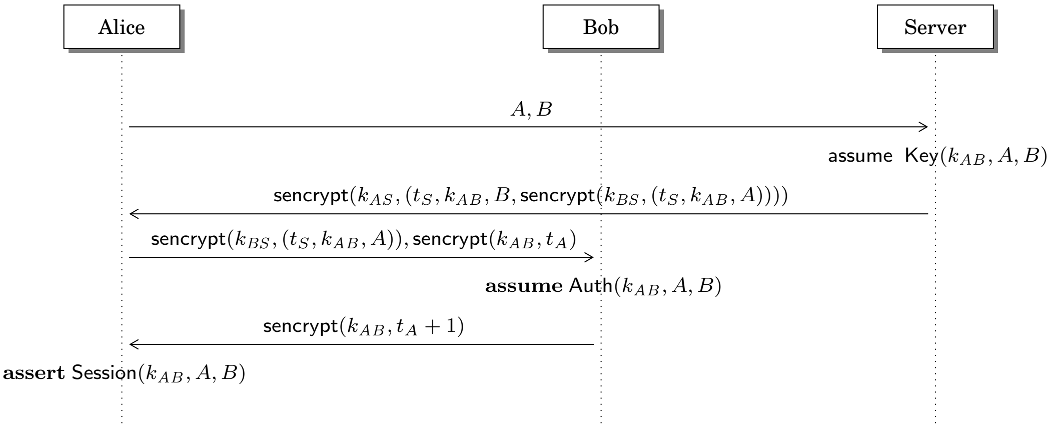

Alice
Bob
Server
A, B
assume Key(k_{AB}, A, B)
sencrypt$(k_{AS}, (t_S, k_{AB}, B,$ sencrypt$(k_{BS}, (t_S, k_{AB}, A))))$
sencrypt$(k_{BS}, (t_S, k_{AB}, A))$, sencrypt$(k_{AB}, t_A)$
assume Auth(k_{AB}, A, B)
sencrypt$(k_{AB}, t_A + 1)$
assert Session(k_{AB}, A, B)

Figure 2.2: The Kerberos protocol (mutual authentication)

The goal of the protocol is to establish a fresh session key k_{AB} between principals A and B through a trusted server S, which shares a symmetric key with both A and B. Kerberos employs timestamps like t_S and t_A to prove session recentness and protect against replay attacks. Initially, A contacts the server S, providing the identities of the two agents A and B who want to establish a session. The server generates a fresh timestamp t_S and a new session key k_{AB}, then it packages all this information into a message for A and a message for B, which are combined by a nested encryption at step 2 of the protocol. Later, A removes the outer layer of the encryption, checks t_S and retrieves k_{AB}. If the timestamp is fresh, she forwards the inner encrypted message to B; additionally, A includes a fresh timestamp t_A encrypted under k_{AB}. Now B can decrypt the message encrypted by S, check its freshness, and retrieve the session key k_{AB}. Using this key, B can disclose the timestamp t_A and reply to A with $t_A + 1$, thus authenticating herself.

2.9.2 Protocol analysis and challenges

An intriguing point for our static verification technique is that the timestamp t_S generated by the server is checked by both A and B to ensure that the session key k_{AB} is fresh. As anticipated, this pattern is more sophisticated than the one we discussed for *EPMO*, but the expressiveness of our underlying affine logic framework allows for a simple encoding, discussed in the next section. For the sake of simplicity, in the following we will just focus on the verification of the initiator A.

We start by defining two predicates used in the analysis: $\mathsf{Key}(k_{AB}, A, B)$ states that k_{AB} is a fresh symmetric key intended to establish a session between A and B, while $\mathsf{Auth}(k_{AB}, A, B)$ formalizes that B wishes to communicate with A using key k_{AB}. Intuitively, these are the guarantees available to A after steps 2 and 4 of the protocol, respectively: by combining these two assurances, A can conclude that k_{AB} is a fresh session key which can be safely used to communicate with B. We model this last information through the predicate $\mathsf{Session}(k_{AB}, A, B)$ and we formalize the previous deduction by assuming the authorization policy:

$$!\forall x, y, z.(\mathsf{Key}(x, y, z) \otimes \mathsf{Auth}(x, y, z) \multimap \mathsf{Session}(x, y, z)).$$

We next discuss how we can show the compliance of the protocol against the previous policy by refinement type-checking.

2.9.3 Implementing and typing timestamps

We turn our attention to the implementation. We build on a very simple library for timestamp management, that we allow the principals to access. We note that timestamps are modeled as monotonic counters. To guarantee the freshness of a timestamp in the case that the opponent executes the protocol function multiple times, we pair the counter with a global, instance-dependent, fresh random bitstring *rand* that is created at the beginning of the protocol specification using the function *mkfresh*. This usage of a random bitstring models the assumption that different sessions of Kerberos running in parallel will use different timestamps. Of course, we could consider more realistic and complicated implementations, but the following one suffices to convey the intuition about our methodology:

```
(* Typing a timestamp *)
type TStamp = TS of (bytes * int)

(* Increment a timestamp by 1 *)
let inc_ts t =
  match t with TS (rt, tt) →
  TS (rt, tt + 1)

(* Pick a fresh timestamp, based on the value stored in r *)
let get_ts r = fun () →
  r := inc_ts !r; !r

(* Check a timestamp t for freshness, based on the value stored in r *)
let check_ts r id t' =
  match !r with TS (rt, tt) →
  match t' with TS (=rt, tt') →
  if (tt' > tt) then
    r := t'; assume F(id, t')
  else
    failwith "not_a_fresh_timestamp"

(* The handle to access the two functions above *)
let init_ts rand glob id =
  let tss = !glob in
  let res = search tss id in
  match res with
  | Some(r) −> (get_ts r, check_ts r id)
  | None    −> let newref = mkref TS (rand, 0) in
                 glob := (id,newref)::tss; (get_ts newref, check_ts newref id)
```

Each principal stores the last received timestamp in a reference, created by an invocation to the function *init_ts*, described below. The function *inc_ts*: TStamp → TStamp is used to increment a timestamp by 1, the function *get_ts*: Ref(TStamp) → unit → TStamp is used to create fresh timestamps, and the dependent function *check_ts*: Ref(TStamp) → x : string → y : TStamp → {_ : unit | F(x, y)} is used to check whether a received timestamp y is fresh and can be used to deem timely a communication with the principal x. The code of the function performs a conditional branch: if the timestamp is fresh, 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 string → α;

as a consequence, *check_ ts* can be given the previous dependent function type, whose refined return type provides the freshness assumption.

The function *init_ ts* is more complicated. It takes three parameters: the global instantiation-specific nonce *rand*, the identity of a principal *id* and a global reference *glob*, containing a list of pairs *(id',r')*, where *id'* is the identity of a principal and *r'* is a reference containing the last timestamp presented by *id'* (*TS (rand, 0)* if none). The function starts by retrieving this list from *glob*, bounding it to *tss*, and then uses an auxiliary function *search* to detect if there exists an entry of the form *(id,r)* in *tss*. If this is the case, *init_ ts* returns a pair of functions *(get_ ts r, check_ ts r id)*, which will allow the caller to get a fresh timestamp and to check the freshness of the timestamps received by *id*. If *id* has never presented a timestamp when *init_ ts* is invoked, the function creates a fresh reference containing *TS (rand, 0)* and updates the list stored in the reference *glob* to preserve the expected invariant, then it returns again a pair of functions for timestamp management. This implementation ensures that different instances of a protocol participant with the same identity will share the same counter for timestamps, which is important to protect the protocol against replay attacks. The *init_ ts* function has type:

$$\begin{aligned}&\mathsf{bytes} \to \mathsf{Ref}(\mathsf{List}(\mathsf{string} * \mathsf{TStamp})) \to x : \mathsf{string} \to\\ &\quad((\mathsf{unit} \to \mathsf{TStamp}) * (y : \mathsf{TStamp} \to \{\_ : \mathsf{unit} \mid \mathsf{F}(x, y)\})).\end{aligned}$$

2.9.4 Typing the session key using self-dependent key types

Before discussing the implementation of the principal A, we must first consider a subtle issue related to verification. We pointed out that, at step 4 of the protocol, A must be able to infer that k_{AB} has been previously authenticated by B. The problem for verification is that the formula $\mathsf{Auth}(k_{AB}, A, B)$ modeling this fact must be conveyed by the type of the key k_{AB} itself, but neither the key k_{AB} nor the two identifiers A and B occur in the *payload* of the last protocol message, hence we cannot predicate on them using dependent typing. While the problem of letting the payload of the key refer to the identifiers A and B can be solved quite easily, since the referred to identities are globally and publicly known, the problem of letting the payload of a key predicate over the key itself is more involved due to lexical scoping. We show how to devise an encoding to solve the problem of self-dependent key types, which is close in spirit to the session key treatment advocated in some of our previous work [2].

Here we rely on a sealing-based encoding, where the *self-dependent* key k_{AB} consists of a *key identifier* i_{AB} and a pair k'_{AB} composed of the sealing and unsealing functions, thus having the form $k_{AB} = (i_{AB}, k'_{AB})$. The predicate Auth of the protocol refers to the identifier i_{AB} of the key k_{AB}, i.e., we actually assume $\mathsf{Auth}(i_{AB}, A, B)$ rather than $\mathsf{Auth}(k_{AB}, A, B)$ as we were discussing in the previous informal overview. The link between each self-dependent key k and its respective key identifier i is logically modeled by the predicate $\mathsf{KeyIdent}(k, i)$, which holds true for all valid key-identifier pairs. The adapted authorization policy then looks as follows:

$$!\forall w, x, y, z.(\mathsf{Key}(x, y, z) \otimes \mathsf{KeyIdent}(x, w) \otimes \mathsf{Auth}(w, y, z) \multimap \mathsf{Session}(x, y, z)).$$

In the following we present the definition of our sealing-based library for the self-dependent session key k_{AB}. For presentation convenience, we make use of the following notation:

$$\textbf{type}\ \mathsf{MsgAB}{<}x{,}y{,}z{>} = (\{t : \mathsf{TStamp} \mid !(\mathsf{F}(y, t) \multimap \mathsf{Auth}(x, y, z))\} + \mathsf{TStamp}$$

to denote the (open) type MsgAB of the session key payload. Here, $x \in fv(\mathsf{MsgAB}{<}x{,}y{,}z{>})$ refers to the key identifier, while $y, z \in fv(\mathsf{MsgAB}{<}x{,}y{,}z{>})$ refer to the globally available public identifiers A and B respectively. Note that this type is a sum type, since the key k_{AB} will be used by B to encrypt a timestamp of type $(t : \mathsf{TStamp})\{!(\mathsf{F}(B, t) \multimap \mathsf{Auth}(x, y, z))\}$ and by A to encrypt a non-refined timestamp of type TStamp (since we do not focus on the verification of B here). The sealing-based library for the dependent key k_{AB} shared between A and B is given below:

```
(∗ Closed type of the session key established by Kerberos.
   Here, w stands for the key identifier discussed above ∗)
type DSymKey = DSym of (w : string ∗ ((MsgAB<w,A,B> → Un) ∗
                                       (Un → MsgAB<w,A,B>)))

(∗ Generate a fresh identifier ∗)
val new_fresh_id: unit → string

(∗ Create a new self-dependent key ∗)
let mkdepkey: unit → DSymKey = fun () →
  let id = new_fresh_id () in
    let s = mkseal "dsymkey" in
      DSym (id,s)

(∗ Get the key identifier corresponding to a self-dependent key ∗)
let get_key_ident k: (k : DSymKey → {x : string | !KeyIdent(k, x)}) =
    match k with DSym (x, _) → assume !KeyIdent(k, x); x

(∗ Self-dependent symmetric encryption function ∗)
let depencrypt x k m: (x : string → DSymKey → MsgAB<x,A,B> → Un) =
    match k with DSym (=x, (seal, _)) → seal m

(∗ Self-dependent symmetric decryption function ∗)
let depdecrypt x k c: (x : string → DSymKey → Un → MsgAB<x,A,B>) =
    match k with DSym (=x, (_, unseal)) → unseal c
```

In the function *mkdepkey* we call the existing seal creation function *mkseal*, which is used to generate a new seal that is paired with a fresh key identifier. Specifically, recall that we have:

```
type Seal(α)  = (α → Un) ∗ (Un → α)
val mkseal: string → Seal(α)
```

In the case of the key generation function *mkdepkey*, the placeholder α is replaced by the monomorphic type MsgAB<*id,A,B*>. Hence we must ensure that *id* is in scope when specializing the *mkseal* function.

Finally, we can briefly comment the other functions of our small library. The function *get_key_ident* extracts the identifier i from a dependent key k and tracks the logical dependence $\mathsf{KeyIdent}(k, i)$ through its refined return type. Contrary to standard sealing-based encryption and decryption, the functions *depencrypt* and *depdecrypt* take the key identifier as an additional argument and perform a pattern-matching operation to bridge the dependent typing allowed by pair splitting and the dependent typing enabled by the definition of these functions. In the syntax of types, the need for this pattern matching operation is made apparent by the occurrence of the same variable x in both the function type of *depencrypt*/*depdecrypt* and the data type MsgAB<*x,A,B*>.

2.9.5 Type-checking the initiator

We finally have all the ingredients to discuss how the initiator A is type-checked. The code of the principal looks as follows:

```
(∗ Authorization policy ∗)
assume !∀w, x, y, z.(Key(x, y, z) ⊗ KeyIdent(x, w) ⊗ Auth(w, y, z) ⊸ Session(x, y, z))

(∗ Typing the message from A to B, where MsgAB<x,y,z> will be
   closed by instantiating it in the definition of the session key type ∗)
type MsgAB<x,y,z> = MsgAB of {t : TStamp | !(F(y, t) ⊸ Auth(x, y, z))} + TStamp

(∗ Typing the session key established by Kerberos ∗)
type DSymKey = DSym of (w : string ∗ ((MsgAB<w,A,B> → Un)∗
                                      (Un → MsgAB<w,A,B>)))

(∗ Typing the message from S to A, where MsgSA<x> will be
   closed by instantiating it in the initiator function ∗)
type MsgSA<x> = MsgSA of (xts : TStamp ∗ xkAB : DSymKey ∗ xB : string ∗ y : Un)
  {!(F(xB, xts) ⊸ Key(xkAB, x, xB))}

(∗ Initiator code, where rand is a fresh global bitstring and
glob denotes a global reference, which are both provided in the
protocol specification and are not under the control of the opponent ∗)
let initiator rand glob A addA B addB S addS (kAS: SymKey(MsgSA<A>)) =
    let (get_tsB, check_tsB) = init_ts rand glob B in
      send addS (A,B);
      let msgSA = receive addA in
      let plainSA = sdecrypt kAS msgSA in
```

```
match plainSA with
  MsgSA (xts, xkAB, =B, y) →
    (* $!(\mathsf{F}(B, xts) \multimap \mathsf{Key}(xkAB, A, B))$ holds true *)
    let _ = check_tsB xts in
    (* $\mathsf{F}(B, xts)$ holds true *)
    let tA = get_tsB () in
    let iAB = get_key_ident xkAB in
    (* $!\mathsf{KeyIdent}(xkAB, iAB)$ holds true *)
    let msgAB = depencrypt iAB xkAB tA in
    send addB (y,msgAB);
    let msgBA = receive addA in
    (* $tA' = tA + 1$ *)
    let tA' = inc_ts tA in
    let (=tA') = depdecrypt iAB xkAB msgBA in
      (* $!(\mathsf{F}(B, tA') \multimap \mathsf{Auth}(iAB, A, B))$ holds true *)
      let _ = check_tsB tA' in
      (* $\mathsf{F}(B, tA')$ holds true *)
      assert $\mathsf{Session}(xkAB, A, B)$
```

Decryption and pattern-matching introduce the guarded formulas needed to type-check the initiator, while invocations to the timestamp library extend the typing environment with the control formulas needed to retrieve the payload formulas of interest. Specifically, the initiator starts by creating the handle to the timestamp library through the call *init_ts B*, which returns the two functions *get_tsB* and *check_tsB*. The interesting point here is the type of *check_tsB*, i.e., $y : \mathsf{TStamp} \rightarrow \{\_ : \mathsf{unit} \mid \mathsf{F}(B, y)\}$, hence a successful call to this function allows for deeming a communication with B as timely. To understand why the function is given that type, recall that *init_ts* has the following type:

$$\begin{aligned}&\mathsf{bytes} \rightarrow \mathsf{Ref}(\mathsf{List}(\mathsf{string} * \mathsf{TStamp})) \rightarrow x : \mathsf{string} \rightarrow \\ &\quad ((\mathsf{unit} \rightarrow \mathsf{TStamp}) * (y : \mathsf{TStamp} \rightarrow \{\_ : \mathsf{unit} \mid \mathsf{F}(x, y)\})).\end{aligned}$$

and observe that *check_tsB* is obtained by projecting the second component of the pair returned by the call *init_ts rand glob B*. Now, the logical environment is populated as follows:

(i) when A decrypts the message from S and performs pattern matching, we introduce the formula $!(\mathsf{F}(B, xts) \multimap \mathsf{Key}(xkAB, A, B))$, based on the type of the symmetric key $kAS : \mathsf{SymKey}(\mathsf{MsgSA}{<}A{>})$;

(ii) when A calls the *check_tsB* function on the timestamp xts received by S, we introduce the formula $\mathsf{F}(B, xts)$, based on the typing discussed above;

(iii) when A calls the *get_key_ident* function on the self-dependent key $xkAB$ shared with B, we introduce the formula $!\mathsf{KeyIdent}(xkAB, iAB)$, where iAB is the key identifier associated to $xkAB$;

(*iv*) when A decrypts the message from B using the self-dependent key $xkAB$ identified by iAB, we introduce the formula $!(\mathsf{F}(B, tA') \multimap \mathsf{Auth}(iAB, A, B))$, based on the type of the *depdecrypt* function associated to $xkAB$, where tA' corresponds to tA incremented by 1;

(*v*) finally, when A calls the *check_ tsB* function on the timestamp tA' received by B, we introduce the formula $\mathsf{F}(B, tA')$, similarly to what we do at point (*ii*).

Using (*i*) and (*ii*), we can prove $\mathsf{Key}(xkAB, A, B)$, while using (*iv*) and (*v*) we can prove $\mathsf{Auth}(iAB, A, B)$. These two formulas, along with $!\mathsf{KeyIdent}(xkAB, iAB)$ at point (*iii*), allow to derive the assertion $\mathsf{Session}(xkAB, A, B)$ based on the underlying authorization policy, hence the initiator is well-typed.

To conclude that the protocol actually respects the authorization policy despite the introduction of the serializers, it is enough to ensure that $\mathsf{F}(B, t)$ is assumed at most once for any possible choice of t. To prove it, we must guarantee that at the beginning of the protocol specification function:

1. the global fresh value *rand* is freshly generated using the function *mkfresh* that never generates the same value twice;

2. the global reference *glob* storing the received timestamps is correctly instantiated to the empty list and is not provided by the opponent as an argument to the protocol specification function. We thus note that different participants running with identity A share the same counter for timestamps management by construction of our library (cf. Section 2.9.3) and that each invocation to *check_ tsB* always returns an assumption predicating over increasing values of t.

2.10 Algorithmic type-checking (AF7$_{\text{alg}}$)

The type system presented in Section 2.6 includes several non-deterministic rules, which make it hard to implement an efficient decision procedure for typing. In this section we outline an algorithmic variant of the type system, which we prove sound and complete. We first focus on presenting the main intuitions behind the algorithmic type system design and then show the complete formalization.

2.10.1 Overview

While standard sources of non-determinism (like subtyping or refining value types) can be eliminated using type annotations, the rewriting of logical environments, which is the distinctive source of non-determinism of our system, is harder to deal with. The core idea underlying the algorithmic version of the type system is to dispense with the logical environment Δ and to construct bottom-up a single logical formula that characterizes all the proof obligations that would normally be introduced along the type derivation. In such a way, all the burden

(Val Var Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \diamond \qquad (x:T)\in\Gamma}{\Gamma \vdash_{\mathsf{alg}} x : T; \mathbf{1}}$$

(Val Fun Alg)

$$\frac{\Gamma, x:\psi(T) \vdash_{\mathsf{alg}} \overline{E} : U; F' \qquad \mathit{fnfv}(T) \subseteq \mathit{dom}(\Gamma)\cup\{x\}}{\Gamma \vdash_{\mathsf{alg}} \lambda x:T.\overline{E} : (x:T\to U); !\forall x.(\mathit{forms}(x:T) \multimap F')}$$

(Val Ref Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \overline{M} : T; F' \qquad \mathit{fnfv}(F)\subseteq \mathit{dom}(\Gamma)\cup\{x\}}{\Gamma\vdash_{\mathsf{alg}} \overline{M}_{\{x:\_ \ \mid\ F\}} : \{x:T \mid F\}; F'\otimes F\{M/x\}}$$

(Val Pair Alg)

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

(Exp Let Alg)

$$\frac{\begin{array}{c}\overline{E} \rightsquigarrow^{\emptyset} [\Delta' \mid \overline{E'}]\\ \Gamma \vdash_{\mathsf{alg}} \overline{E'} : T; F_1 \qquad \Gamma, x:\psi(T)\vdash_{\mathsf{alg}} \overline{D} : U; F_2 \qquad x\notin \mathit{fv}(U) \qquad \mathit{fnfv}(\Delta')\subseteq \mathit{dom}(\Gamma)\end{array}}{\Gamma\vdash_{\mathsf{alg}} \mathsf{let}\ x=\overline{E}\ \mathsf{in}\ \overline{D} : U; \Delta' \multimap (F_1\otimes\forall x.(\mathit{forms}(x:T)\multimap F_2))}$$

Notation: Here $E := \langle\overline{E}\rangle$ denotes the expression obtained from $\overline{E}$ by erasing all its typing annotations.

Table 2.14: Selected algorithmic rules for typing values and expressions (AF7$_{\text{alg}}$)

due to resource management can be shifted to an external affine logic theorem prover, which has to deal with this issue anyway.

More in detail, every typing judgement of the form $\Gamma;\Delta\vdash\mathcal{J}$ is matched by an algorithmic counterpart of the form $\Gamma\vdash_{\mathsf{alg}}\mathcal{J};F$. Intuitively, typing an expression algorithmically 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 fully deterministic, and in case of success yields *one* proof obligation F.

2. The proof obligation is verified, e.g., using an external theorem prover.

If both steps succeed, then the expression is well-typed.

2.10.2 Key ideas

We illustrate the main ideas behind our algorithmic type system on some representative rules, shown in Table 2.14. The algorithmic rules for kinding (cf. Section 2.10.4), subtyping (cf. Section 2.10.5), and typing the remaining values and expressions (cf. Section 2.10.6) follow along the same lines. For the sake of readability we often abuse notation and we let the multiset $F_1,\ldots,F_n$ stand for the formula $F_1\otimes\ldots\otimes F_n$.

We first notice that, according to standard practice, we rely on typing annotations to deal with non-structural rules. Annotated terms and expressions are denoted by $\overline{M}$ and $\overline{E}$, respectively. Their syntax is given in Table 2.19. The explicit erasure of all typing annotations of an expression is denoted by $\langle\overline{E}\rangle$. For instance, we explicitly annotate values that are expected to be given a refinement type (cf. (Val Ref Alg)) with the expected refinement F and use annotations to assign an explicit argument type T to functional values (cf. (Val Fun Alg)). In this way, every possible syntactic form for expressions is matched by a single type rule and the selection of appropriate types and refinements does not rely on non-determinism.

We now exemplify the general concepts underlying our technique by contrasting the standard typing rule (Val Fun) with its algorithmic counterpart (Val Fun Alg). The main source of non-determinism in (Val Fun) is the rewriting of Δ to $!\Delta'$. As previously mentioned, our goal is to dispense with logical environments and their rewriting, by collecting a single proof obligation that accounts for the proof obligations generated in the original type system. In the algorithmic version, the proof obligation obtained by giving $\lambda x : T.\, \overline{E}$ type $V = x : T \to U$ in the environment Γ is:

$$!\forall x.(\mathit{forms}(x : T) \multimap F'),$$

where F' is the proof obligation collected by giving $\overline{E}$ type U in $\Gamma, x : \psi(T)$.

In the following, we briefly justify why this approach is sound, i.e., we argue why $\Gamma \vdash_{\mathsf{alg}} \lambda x : T.\, \overline{E} : V; !\forall x.(\mathit{forms}(x : T) \multimap F')$ implies that $\Gamma; \Delta \vdash \lambda x.\, \langle\overline{E}\rangle : V$ for any Δ such that $\Gamma; \Delta \vdash !\forall x.(\mathit{forms}(x : T) \multimap F')$. Notice that the latter judgement is equivalent to assuming that Δ entails $!\forall x.(\mathit{forms}(x : T) \multimap F')$ and both the multiset and the formula are well-formed with respect to Γ. Using the rules of the logic, we can show that a proof of $\Gamma; \Delta \vdash !\forall x.(\mathit{forms}(x : T) \multimap F')$ implies that there exists Δ' such that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$ and:

$$\Gamma; !\Delta' \vdash \forall x.\mathit{forms}(x : T) \multimap F'.$$

Intuitively, this means that we can eliminate the exponential modality by rewriting the logical environment in exponential form. Furthermore, the well-formedness of the (algorithmic) environment $\Gamma, x : \psi(T)$ and the (non-algorithmic) environment $\Gamma; !\Delta'$ ensures that $x \notin \mathit{dom}(\Gamma)$ and thus $x \notin \mathit{fv}(!\Delta')$: in this case, the logic allows us to further eliminate the universal quantification, adding a type binding for x in order to keep the logical environment well-formed (the actual type is not relevant from the logic point of view). Thus, we have:

$$\Gamma, x : \psi(T); !\Delta' \vdash \mathit{forms}(x : T) \multimap F'.$$

Using rule ($\multimap$-Left), we can finally prove:

$$\Gamma, x : \psi(T); !\Delta', \mathit{forms}(x : T) \vdash F'.$$

By inductive reasoning, $\Gamma, x : \psi(T); !\Delta', \mathit{forms}(x : T) \vdash \langle\overline{E}\rangle : U$, hence (Val Fun) allows us to derive $\Gamma; \Delta \vdash \lambda x.\, \langle\overline{E}\rangle : V$. The proof of completeness is similar.

The other algorithmic (typing) rules are constructed along the same lines, using the following additional observations:

- If a typing rule contains no kinding, subtyping, or typing premise (e.g., (Val Var)), the proof obligation of the corresponding algorithmic rule is set to **1** (cf. (Val Var Alg)) and thus trivially fulfilled.
- If a typing rule contains multiple premises (e.g., (Val Pair)), then we combine the proof obligations obtained from the premises conjunctively (cf. (Val Pair Alg)).
- If a typing rule relies on extraction (e.g., (Exp Let)) and adds the extracted environment Δ' to the environment before rewriting, the algorithmic variant of the rule (Exp Let Alg) creates a proof obligation of the form $\Delta' \multimap F$, where F is the proof obligation obtained by combining the proof obligations of the premises using the techniques described above.

With these insights in mind, we now show the complete formalization of the algorithmic type system.

2.10.3 Base judgements

The base judgements of the algorithmic type system are reported in Table 2.10.3.

(Env Empty Alg)
$$\varepsilon \vdash_{\mathsf{alg}} \diamond$$

(Type Env Entry Alg)
$$\frac{\Gamma \vdash_{\mathsf{alg}} \diamond \qquad dom(\mu) \cap dom(\Gamma) = \emptyset \\ \mu = x : T \Rightarrow T = \psi(T) \wedge \mathit{fnfv}(T) \subseteq dom(\Gamma)}{\Gamma, \mu \vdash_{\mathsf{alg}} \diamond}$$

(Type Alg)
$$\frac{\Gamma \vdash_{\mathsf{alg}} \diamond \qquad \mathit{fnfv}(T) \subseteq dom(\Gamma)}{\Gamma \vdash_{\mathsf{alg}} T}$$

Table 2.15: Algorithmic well-formedness judgements (AF7$_{\text{alg}}$)

The only remarkable point here is that we do not have any algorithmic counterpart of rule (Derive). In fact, we never need to prove a formula in the algorithmic formulation of the type system, but we just collect the proof obligation for the external affine logic theorem prover.

2.10.4 Kinding

Table 2.16 presents the algorithmic kinding rules. The non-inductive standard kinding rules (Kind Var) and (Kind Unit), which just check well-formedness of the environment (or environment membership) and which do not contain a proof obligation of the form $\Gamma; \Delta \vdash F$ amongst their hypotheses, are translated into algorithmic rules that generate the proof obligation $\mathbf{1}$. All other (recursive) rules (e.g., (Kind Fun)) strongly resemble their algorithmic counterparts (e.g., (Kind Fun Alg)). The proof obligation that is generated in the algorithmic variant consists of a conjunction of the proof obligations that are recursively generated by the premises of that rule, following the same principles of the algorithmic typing rules for values and expressions that we discussed in Section 2.10.2. Note that a premise that checks the well-formedness of an environment or type does not generate a proof obligation (cf. (Kind Refine Public Alg)).

(Kind Var Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \diamond \qquad (\alpha :: k) \in \Gamma}{\Gamma \vdash_{\mathsf{alg}} \alpha :: k; \mathbf{1}}$$

(Kind Unit Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \diamond}{\Gamma \vdash_{\mathsf{alg}} \mathsf{unit} :: k; \mathbf{1}}$$

(Kind Fun Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} T :: \overline{k}; F_1 \qquad \Gamma, x : \psi(T) \vdash_{\mathsf{alg}} U :: k; F_2}{\Gamma \vdash_{\mathsf{alg}} x : T \to U :: k; !F_1 \otimes !F_2}$$

(Kind Pair Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} T :: k; F_1 \qquad \Gamma, x : \psi(T) \vdash_{\mathsf{alg}} U :: k; F_2}{\Gamma \vdash_{\mathsf{alg}} x : T * U :: k; !F_1 \otimes !F_2}$$

(Kind Sum Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} T :: k; F_1 \qquad \Gamma \vdash_{\mathsf{alg}} U :: k; F_2}{\Gamma \vdash_{\mathsf{alg}} T + U :: k; !F_1 \otimes !F_2}$$

(Kind Rec Alg)

$$\frac{\Gamma, \alpha :: k \vdash_{\mathsf{alg}} T :: k; F}{\Gamma \vdash_{\mathsf{alg}} \mu\alpha.T :: k; !F}$$

(Kind Refine Public Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \{x : T \mid F\} \qquad \Gamma \vdash_{\mathsf{alg}} T :: \mathsf{pub}; F'}{\Gamma \vdash_{\mathsf{alg}} \{x : T \mid F\} :: \mathsf{pub}; F'}$$

(Kind Refine Tainted Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \psi(T) :: \mathsf{tnt}; F' \qquad \Gamma, x : \psi(T) \vdash_{\mathsf{alg}} \diamond \qquad T \text{ refined}}{\Gamma \vdash_{\mathsf{alg}} \{x : T \mid F\} :: \mathsf{tnt}; (\forall x.\mathit{forms}(x : T)) \otimes F'}$$

Table 2.16: Algorithmic kinding relation ($\mathrm{AF7}_{\mathrm{alg}}$)

2.10.5 Subtyping

The algorithmic subtyping rules are presented in Table 2.18. They resolve the non-determinism related to the environment splitting by following the key insights of algorithmic typing presented in Section 2.10.2.

Furthermore, the algorithmic subtyping rules resolve the non-determinism that arises due to the fact that standard subtyping is not syntax-driven as described in the following. We use $T \neq_{\top} U$ to denote that T and U are not refined and do not share the same top-level constructor.

The algorithmic type system makes use of the following observation: for all non-refined types T, U there are at most three standard subtyping rules applicable, namely (Sub Refl), (Sub Pub Tnt), and in the case that T and U share the same top-level constructor one corresponding structural subtyping rule, e.g., (Sub Fun) or (Sub Pair). In the case that T or U are refined, the three standard subtyping rules (Sub Refl), (Sub Pub Tnt), or (Sub Refine) might be applicable.

To reduce this level of non-determinism the algorithmic subtyping rules allow the reflexivity rule (Sub Refl Alg) to be applied only to the non-inductive type unit and type variables α. Furthermore, we restrict the application of the kinding based rule (Sub Pub Tnt Alg) to types T, U that are structurally different and not refined, i.e., $T \neq_{\top} U$. Therefore, we can determine the appropriate subtyping rule by simple syntactic checks. Note that two types T, U, which share the same top-level constructor can still be subtyped using reflexivity or kinding by recursively applying the corresponding structural subtyping rule until one of the subgoals matches the premise of either the (Sub Refl Alg) or (Sub Pub Tnt Alg) rule. Similarly, if either T or U or both are refined they can be typed using reflexivity or kinding by first applying the refinement rule (Sub Refine Alg) and then applying either the (Sub Refl Alg) or (Sub Pub Tnt Alg) rule to the subgoal.

This approach is sound and complete for all but the subtyping of two iso-recursive types. This is related to our choice of adapting the iso-recursive subtyping proposed by Backes et al. [32, 33], which requires the recursive variable to occur only positively in the iso-recursive type, instead of the Amber rule (cf. (Sub Pos Rec) in Section 2.6.4). For instance, given the above constraints, subtyping $\Gamma \vdash_{\mathsf{alg}} \mu\alpha.\,(x : \alpha \to T) <: \mu\alpha.\,(x : \alpha \to T); F$ or $\Gamma \vdash_{\mathsf{alg}} \mu\alpha.\,(x : \alpha \to \mathsf{unit}) <: \mu\alpha.\,(x : \alpha \to \mathsf{unit} + \mathsf{unit}); F$ would not be possible, thus lacking reflexivity and kinding based algorithmic subtyping for iso-recursive types. Therefore, our algorithmic type system contains three rules for subtyping two iso-recursive types: (Sub Refl Rec Alg), (Sub Pub Tnt Rec Alg), and (Sub Pos Rec Alg), respectively. While checking whether or not to apply rule (Sub Refl Rec Alg) can be done by performing a simple equality check on the types, the decision between (Sub Pub Tnt Rec Alg) and (Sub Pos Rec Alg) requires some guidance, leading to the introduction of manual annotations of the form SPT to denote that the rule (Sub Pub Tnt Rec Alg) should be applied. This annotation appears in the subtyping rule for expressions (cf. (Exp Subsum Alg)), which we explain in Section 2.10.6.

The syntax of annotated types $\overline{T}$ is introduced in Table 2.17. Intuitively, we allow type annotations SPT only on iso-recursive types and require them to not be nested. We let $\langle\overline{T}\rangle$ denote the explicit erasure of all annotations SPT from an annotated type $\overline{T}$. To facilitate readability we often write T to denote the non-annotated counterpart $\langle\overline{T}\rangle$ of the annotated type $\overline{T}$. We can easily extend the definition of the function ψ (used for the removal of top-level refinements) to annotated types.

2.10.6 Typing values and expressions

The algorithmic typing rules for values and expressions are given in Table 2.10.6 and Table 2.10.6, respectively. The rules follow according to the intuition de-

$$\begin{array}{lll}
\overline{T}, \overline{U}, \overline{V} ::= & & \textit{annotated types} \\
& \mathsf{unit} & \text{unit} \\
& \alpha & \text{type variable} \\
& x : \overline{T} \to \overline{U} & \text{dependent function type (scope of } x \text{ is } \overline{U}\text{)} \\
& x : \overline{T} * \overline{U} & \text{dependent pair type (scope of } x \text{ is } \overline{U}\text{)} \\
& \overline{T} + \overline{U} & \text{sum type} \\
& \mu\alpha.\overline{T} & \text{iso-recursive type without top-level annotation} \\
& (\mu\alpha.T)_{\mathsf{SPT}} & \text{iso-recursive type with top-level annotation} \\
& \{x : \overline{T} \mid F\} & \text{refinement type (scope of } x \text{ is } F\text{)}
\end{array}$$

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

$$\langle\overline{U}\rangle = \begin{cases} \mathsf{unit} & \text{if } \overline{U} = \mathsf{unit} \\ \alpha & \text{if } \overline{U} = \alpha \\ x : \langle\overline{U_1}\rangle \to \langle\overline{U_2}\rangle & \text{if } \overline{U} = x : \overline{U_1} \to \overline{U_2} \\ x : \langle\overline{U_1}\rangle * \langle\overline{U_2}\rangle & \text{if } \overline{U} = x : \overline{U_1} * \overline{U_2} \\ \langle\overline{U_1}\rangle + \langle\overline{U_2}\rangle & \text{if } \overline{U} = \overline{U_1} + \overline{U_2} \\ \mu\alpha.\,\langle\overline{T}\rangle & \text{if } \overline{U} = \mu\alpha.\overline{T} \\ \mu\alpha.T & \text{if } \overline{U} = (\mu\alpha.T)_{\mathsf{SPT}} \\ \{x : \langle\overline{T}\rangle \mid F\} & \text{if } \overline{U} = \{x : \overline{T} \mid F\} \end{cases}$$

Table 2.17: Syntax of annotated AF7_{alg} types and annotation erasure

scribed in Section 2.10.2. We furthermore rely on type annotations to guide the selection of applicable typing rules and appropriate types. The syntax of annotated values and expressions is given in Table 2.19. Here "_" is used to denote a type that is derived by the typing rules and thus does not need to be specified by the annotator. We denote the recursive erasure of all typing annotations by $\langle\overline{E}\rangle$ and often use E to denote the expression $\langle\overline{E}\rangle$ obtained from the annotated expression $\overline{E}$ by erasing all its typing annotations. The extraction relation $\overline{E} \rightsquigarrow^{\emptyset} [\Delta \mid \overline{D}]$ for annotated expressions (cf. Table 2.10.6) extracts formulas as in the non-annotated case while keeping annotations on the expressions intact but for the case of assumptions, where it changes the type annotation in the original assumption to a subtyping annotation in the extracted assumption for all types different from unit. The notions of free names and free variables correspond to the non-annotated case.

Since in the typing rule (Val Fun) for functions the type of the input is chosen non-deterministically, we use the annotation $\lambda x : T.\,E$ to guide the algorithmic type system (Val Fun Alg) in the selection of a suitable input type T. The annotation $M_{\{x:\_ \mid F\}}$ explicitly triggers the rule (Val Ref Alg) and

(Sub Refl Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} T \qquad T \in \{\mathsf{unit}, \alpha\}}{\Gamma \vdash_{\mathsf{alg}} T <: T; \mathbf{1}}$$

(Sub Pub Tnt Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} T :: \mathsf{pub}; F_1 \qquad \Gamma \vdash_{\mathsf{alg}} U :: \mathsf{tnt}; F_2 \qquad T \neq_{\top} U}{\Gamma \vdash_{\mathsf{alg}} T <: U; F_1 \otimes F_2}$$

(Sub Fun Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \overline{T'} <: \overline{T}; F_1 \qquad \Gamma, x : \psi(T') \vdash_{\mathsf{alg}} \overline{U} <: \overline{U'}; F_2}{\Gamma \vdash_{\mathsf{alg}} x : \overline{T} \to \overline{U} <: x : \overline{T'} \to \overline{U'}; !F_1 \otimes !F_2}$$

(Sub Pair Alg)

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{alg}} \overline{T} <: \overline{T'}; F_1 \\ \Gamma, x : \psi(T) \vdash_{\mathsf{alg}} \overline{U} <: \overline{U'}; F_2\end{array}}{\Gamma \vdash_{\mathsf{alg}} x : \overline{T} * \overline{U} <: x : \overline{T'} * \overline{U'}; !F_1 \otimes !F_2}$$

(Sub Sum Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \overline{T} <: \overline{T'}; F_1 \qquad \Gamma \vdash_{\mathsf{alg}} \overline{U} <: \overline{U'}; F_2}{\Gamma \vdash_{\mathsf{alg}} \overline{T} + \overline{U} <: \overline{T'} + \overline{U'}; !F_1 \otimes !F_2}$$

(Sub Pos Rec Alg)

$$\frac{\Gamma, \alpha \vdash_{\mathsf{alg}} T <: T'; F \qquad \overline{T} \neq \overline{T'} \qquad \alpha \text{ occurs only positively in } \overline{T} \text{ and } \overline{T'}}{\Gamma \vdash_{\mathsf{alg}} \mu\alpha.\,\overline{T} <: \mu\alpha.\,\overline{T'}; !F}$$

(Sub Refl Rec Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \mu\alpha.\,T}{\Gamma \vdash_{\mathsf{alg}} \mu\alpha.\,T <: \mu\alpha.\,T; \mathbf{1}}$$

(Sub Pub Tnt Rec Alg)

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{alg}} \mu\alpha.\,T :: \mathsf{pub}; F_1 \qquad \Gamma \vdash_{\mathsf{alg}} \mu\alpha.\,U :: \mathsf{tnt}; F_2 \\ s = \mathsf{SPT} \oplus s' = \mathsf{SPT}\end{array}}{\Gamma \vdash_{\mathsf{alg}} (\mu\alpha.\,T)_s <: (\mu\alpha.\,T')_{s'}; F_1 \otimes F_2}$$

(Sub Refine Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \psi(\overline{T}) <: \psi(\overline{U}); F \qquad \overline{T} \text{ and/or } \overline{U} \text{ refined} \qquad \Gamma \vdash_{\mathsf{alg}} T \qquad \Gamma \vdash_{\mathsf{alg}} U}{\Gamma \vdash_{\mathsf{alg}} \overline{T} <: \overline{U}; F \otimes \forall y.(\mathit{forms}(y : T) \multimap \mathit{forms}(y : U))}$$

Notation: We write $T \neq_{\top} U$ to denote that T and U are not refined and do not share the same top-level constructor. $\oplus$ denotes the exclusive or. We use T to denote the non-annotated counterpart $\langle\overline{T}\rangle$ of the annotated type $\overline{T}$.

Table 2.18: Algorithmic subtyping relation ($\mathrm{AF7_{alg}}$)

expects M to type-check with refinement F, while the annotations $(\mathsf{inl}\ M)_{\_+U}$ and $(\mathsf{inl}\ M)_{T+\_}$ are used to provide the respective missing type in the sum type $T + U$ that will be assigned to $\mathsf{inl}\ M$ and $\mathsf{inr}\ M$ (cf. (Val Inl Alg) and (Val

Inr Alg)). Furthermore, the rule (Exp Subsum) is highly non-deterministic, since its application can be tried at any time using any combination of possible sub- and supertypes. In the algorithmic version of the type system we prevent the unnecessary application of subtyping and help the choice of an appropriate supertype T' by annotating an expression $\overline{E}$ as $\overline{E}_{\_<:\overline{T'}}$ whenever subtyping is necessary (cf. (Exp Subsum Alg)). Note that the type T' will additionally be annotated with SPT in case that the subtyping should make use of rule (Sub Pub Tnt Rec Alg), resulting in the annotated type $\overline{T'}$. Since the typing rule (Exp Assume) non-deterministically chooses a type T, its algorithmic counterpart (Exp Assume Alg) requires an explicit annotation of the form $(\mathsf{assume}\ F)_T$ to provide the expected type T.

$\overline{M}, \overline{N}$ ::=		*values*
	x	variable
	$()$	unit
	$(\overline{M}, \overline{N})$	pair
	$\lambda x : T.\, \overline{E}$	annotated function with input of type T
	$(\mathsf{inl}\ \overline{M})_{\_+T}$	annotated left constructor
	$(\mathsf{inr}\ \overline{M})_{T+\_}$	annotated right constructor
	$\mathsf{fold}\ \overline{M}$	fold constructor
	$\overline{M}_{\{x:\_ \mid F\}}$	value to be refined with F
	$\overline{M}_{\_<:\overline{T}}$	value to be subtyped to $\overline{T}$
$\overline{D}, \overline{E}$ ::=		*expressions*
	$\overline{M}$	value
	$\overline{M}\ \overline{N}$	application
	$\overline{M} = \overline{N}$	syntactic equality
	$\mathsf{let}\ x = \overline{E}\ \mathsf{in}\ \overline{E}'$	let (scope of x is $\overline{E'}$)
	$\mathsf{let}\ (x, y) = \overline{M}\ \mathsf{in}\ \overline{E}$	pair split (scope of x, y is $\overline{E}$)
	$\mathsf{match}\ \overline{M}\ \mathsf{with}\ h\ x\ \mathsf{then}\ \overline{E}\ \mathsf{else}\ \overline{E'}$	match (scope of x is $\overline{E}$)
	$(\nu a)\overline{E}$	restriction (scope of a is $\overline{E}$)
	$\overline{E} \Rsh \overline{E'}$	fork
	$a!\overline{M}$	message send
	$a?$	message receive
	$\mathsf{assume}\ 1$	non-annotated truth assumption
	$(\mathsf{assume}\ F)_T$	annotated assumption with expected type T
	$\mathsf{assert}\ F$	assertion
	$\overline{E}_{\_<:\overline{T}}$	expression to be subtyped to $\overline{T}$

Table 2.19: Syntax of annotated RCF_{AF7} expressions

2.10.7 Formal results

We can state and prove the following formal results, which highlight the correctness and the accuracy of the algorithmic type system.

(Val Var Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \diamond \qquad (x : T) \in \Gamma}{\Gamma \vdash_{\mathsf{alg}} x : T; \mathbf{1}}$$

(Val Unit Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \diamond}{\Gamma \vdash_{\mathsf{alg}} () : \mathsf{unit}; \mathbf{1}}$$

(Val Fun Alg)

$$\frac{\Gamma, x : \psi(T) \vdash_{\mathsf{alg}} \overline{E} : U; F' \qquad \mathit{fnfv}(T) \subseteq \mathit{dom}(\Gamma) \cup \{x\}}{\Gamma \vdash_{\mathsf{alg}} \lambda x : T.\, \overline{E} : x : T \to U; !\forall x.(\mathit{forms}(x : T) \multimap F')}$$

(Val Pair Alg)

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

(Val Ref Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \overline{M} : T; F' \qquad \mathit{fnfv}(F) \subseteq \mathit{dom}(\Gamma) \cup \{x\}}{\Gamma \vdash_{\mathsf{alg}} \overline{M}_{\{x:\_ \ \mid\ F\}} : \{x : T \mid F\}; F' \otimes F\{M/x\}}$$

(Val Inl Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \overline{M} : T; F' \qquad \Gamma \vdash_{\mathsf{alg}} U}{\Gamma \vdash_{\mathsf{alg}} (\mathsf{inl}\ \overline{M})_{\_+U} : T + U; !F'}$$

(Val Inr Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \overline{M} : U; F' \qquad \Gamma \vdash_{\mathsf{alg}} T}{\Gamma \vdash_{\mathsf{alg}} (\mathsf{inr}\ \overline{M})_{T+\_} : T + U; !F'}$$

(Val Fold Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \overline{M} : T\{\mu a.T/\alpha\}; F'}{\Gamma \vdash_{\mathsf{alg}} \mathsf{fold}\ \overline{M} : \mu\alpha.T; !F'}$$

Notation: Here $M = \langle \overline{M} \rangle$ denotes the value obtained from $\overline{M}$ by erasing all its typing annotations.

Table 2.20: Algorithmic typing of values (AF7$_{\text{alg}}$)

Theorem 2.4 (Soundness of Algorithmic Typing)**.** *If* $\Gamma \vdash_{\mathsf{alg}} \overline{E} : T; F$ *and* $\Gamma; \Delta \vdash F$*, then* $\Gamma; \Delta \vdash \langle \overline{E} \rangle : T$.

Proof. See Appendix A.2. □

Theorem 2.5 (Completeness of Algorithmic Typing)**.** *If* $\Gamma; \Delta \vdash E : T$*, then there exist* $\overline{E}, F$ *such that* $\langle \overline{E} \rangle = E$ *and* $\Gamma \vdash_{\mathsf{alg}} \overline{E} : T; F$ *and* $\Gamma; \Delta \vdash F$.

Proof. See Appendix A.2. □

2.10.8 Example

The proof obligation assigned to the *cust* function in Section 2.8 by the algorithmic formulation of our type system is shown below:

$$\begin{array}{l}
\forall C.\forall M.\forall B.\forall g.\forall p. \\
\quad \forall nC.((\mathsf{N1}(nC) \otimes \mathsf{N2}(nC)) \multimap \\
\qquad \forall xnM.(!(\mathsf{N1}(nC) \multimap (\forall y.\mathsf{Pay}(y, p, M, xnM) \multimap \mathsf{Ship}(M, g, C))) \multimap \\
\qquad\qquad\quad !(\mathsf{N2}(nC) \multimap (\forall z.\mathsf{Pay}(B, p, z, xnM))) \multimap \\
\qquad\qquad\qquad\quad \mathsf{Ship}(M, g, C)))
\end{array}$$

(Exp Subsum Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \overline{E} : T; F_1 \qquad \Gamma \vdash_{\mathsf{alg}} T <: \overline{T'}; F_2}{\Gamma \vdash_{\mathsf{alg}} \overline{E}_{\_<:\overline{T'}} : T'; F_1 \otimes F_2}$$

(Exp Appl Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \overline{M} : x : T \to U; F_1 \qquad \Gamma \vdash_{\mathsf{alg}} \overline{N} : T; F_2}{\Gamma \vdash_{\mathsf{alg}} \overline{M}\ \overline{N} : U\{N/x\}; F_1 \otimes F_2}$$

(Exp Let Alg)

$$\frac{\overline{E} \leadsto^{\emptyset} [\Delta' \mid \overline{E'}] \qquad \Gamma \vdash_{\mathsf{alg}} \overline{E'} : T; F_1 \qquad \Gamma, x : \psi(T) \vdash_{\mathsf{alg}} \overline{D} : U; F_2 \qquad x \notin \mathit{fv}(U) \qquad \mathit{fnfv}(\Delta') \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash_{\mathsf{alg}} \mathsf{let}\ x = \overline{E}\ \mathsf{in}\ \overline{D} : U; \Delta' \multimap (F_1 \otimes \forall x.(\mathit{forms}(x : T) \multimap F_2))}$$

(Exp Split Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \overline{M} : x : T * U; F_1 \qquad \Gamma, x : \psi(T), y : \psi(U) \vdash_{\mathsf{alg}} \overline{E} : V; F_2 \qquad \{x, y\} \cap \mathit{fv}(V) = \emptyset}{\Gamma; \Delta \vdash_{\mathsf{alg}} \mathsf{let}\ (x, y) = \overline{M}\ \mathsf{in}\ \overline{E} : V; \quad F_1 \otimes \forall x.\forall y.(\mathit{forms}(x : T) \otimes \mathit{forms}(y : U) \otimes !((x, y) = M) \multimap F_2)}$$

(Exp Match Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \overline{M} : T; F_1 \qquad \Gamma, x : \psi(H) \vdash_{\mathsf{alg}} \overline{E} : U; F_2 \qquad \Gamma; \Delta_2 \vdash_{\mathsf{alg}} \overline{D} : U; F_3 \qquad (h, H, T) \in \{(\mathsf{inl}, T_1, T_1 + T_2), (\mathsf{inr}, T_2, T_1 + T_2), (\mathsf{fold}, T'\{\mu\alpha.T'/\alpha\}, \mu\alpha.T')\} \qquad \mathit{fnfv}(H) \subseteq \mathit{dom}(\Gamma) \cup \{x\}}{\Gamma; \Delta \vdash_{\mathsf{alg}} \mathsf{match}\ \overline{M}\ \mathsf{with}\ h\ x\ \mathsf{then}\ \overline{E}\ \mathsf{else}\ \overline{D} : U; \quad F_1 \otimes \forall x.(\mathit{forms}(x : H) \otimes !(h\ x = M) \multimap F_2) \otimes F_3}$$

(Exp Eq Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \overline{M} : T; F_1 \qquad \Gamma \vdash_{\mathsf{alg}} \overline{N} : U; F_2 \qquad x \notin (\mathit{fv}(\overline{M}) \cup \mathit{fv}(\overline{N}))}{\Gamma \vdash_{\mathsf{alg}} \overline{M} = \overline{N} : \{x : \mathsf{bool} \mid !(x = \mathsf{true} \multimap M = N)\}; F_1 \otimes F_2}$$

(Exp True Alg)

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

(Exp Assume Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} (\mathsf{assume}\ \mathbf{1})_{\_<:T} : T; F' \qquad F \neq \mathbf{1} \qquad \mathit{fnfv}(F) \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash_{\mathsf{alg}} (\mathsf{assume}\ F)_T : T; F \multimap F'}$$

(Exp Assert Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \diamond \qquad \mathit{fnfv}(F) \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash_{\mathsf{alg}} \mathsf{assert}\ F : \mathsf{unit}; F}$$

(Exp Res Alg)

$$\frac{\overline{E} \leadsto^{a} [\Delta' \mid \overline{E'}] \qquad \Gamma, a \updownarrow T \vdash_{\mathsf{alg}} \overline{E'} : U; F \qquad a \notin \mathit{fn}(U) \qquad \mathit{fnfv}(\Delta') \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash_{\mathsf{alg}} (\nu a \updownarrow T)\overline{E} : U; \Delta' \multimap F}$$

(Exp Send Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \overline{M} : T; F \qquad (a \updownarrow T) \in \Gamma}{\Gamma \vdash_{\mathsf{alg}} a!\overline{M} : \mathsf{unit}; F}$$

(Exp Recv Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \diamond \qquad (a \updownarrow T) \in \Gamma}{\Gamma \vdash_{\mathsf{alg}} a? : T; \mathbf{1}}$$

(Exp Fork Alg)

$$\frac{\overline{E_1} \leadsto^{\emptyset} [\Delta_1 \mid \overline{D_1}] \qquad \overline{E_2} \leadsto^{\emptyset} [\Delta_2 \mid \overline{D_2}] \qquad \Gamma \vdash_{\mathsf{alg}} \overline{D_1} : T_1; F_A \qquad \Gamma \vdash_{\mathsf{alg}} \overline{D_2} : T_2; F_B \qquad \mathit{fnfv}(\Delta_1, \Delta_2) \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash_{\mathsf{alg}} \overline{E_1} \upharpoonright \overline{E_2} : T_2; (\Delta_1, \Delta_2) \multimap (F_A \otimes F_B)}$$

Notation: Here $E = \langle \overline{E} \rangle$ denotes the expression obtained from $\overline{E}$ by erasing all its typing annotations.

Table 2.21: Algorithmic typing of expressions (AF7$_{\text{alg}}$)

For the sake of readability we removed all unnecessary occurrences of **1** and all 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

(Extr Fork)
$$\dfrac{\overline{E_1} \leadsto^{\widetilde{a}} [\Delta_1 \mid \overline{D_1}] \qquad \overline{E_2} \leadsto^{\widetilde{a}} [\Delta_2 \mid \overline{D_2}]}{(\overline{E_1} \Rsh \overline{E_2})_s \leadsto^{\widetilde{a}} [\Delta_1, \Delta_2 \mid (\overline{D_1} \Rsh \overline{D_2})_s]}$$

(Extr Let)
$$\dfrac{\overline{E_1} \leadsto^{\widetilde{a}} [\Delta \mid \overline{D_1}]}{(\mathsf{let}\ x = \overline{E_1}\ \mathsf{in}\ \overline{E_2})_s \leadsto^{\widetilde{a}} [\Delta \mid (\mathsf{let}\ x = \overline{D_1}\ \mathsf{in}\ \overline{E_2})_s]}$$

(Extr Res)
$$\dfrac{\overline{E} \leadsto^{a,\widetilde{b}} [\Delta \mid \overline{D}]}{((\nu a)\overline{E})_s \leadsto^{\widetilde{b}} [\Delta \mid ((\nu a)\overline{D})_s]}$$

(Extr Assume Unit)
$$\dfrac{F \neq \mathbf{1} \qquad \mathit{fn}(F) \cap \{\widetilde{a}\} = \emptyset}{((\mathsf{assume}\ F)_{\mathsf{unit}})_s \leadsto^{\widetilde{a}} [F \mid (\mathsf{assume}\ \mathbf{1})_s]}$$

(Extr Assume)
$$\dfrac{F \neq \mathbf{1} \qquad \mathit{fn}(F) \cap \{\widetilde{a}\} = \emptyset \qquad T \neq \mathsf{unit}}{((\mathsf{assume}\ F)_T)_s \leadsto^{\widetilde{a}} [F \mid ((\mathsf{assume}\ \mathbf{1})_{\_<:T})_s]}$$

(Extr Exp)
$$\dfrac{\text{no other rule applies}}{\overline{E} \leadsto^{\widetilde{a}} [\emptyset \mid \overline{E}]}$$

Remark: Note that here s is either SPT or ϵ (i.e., no annotation).

Table 2.22: The extraction relation for annotated expressions ($\mathrm{AF7_{alg}}$)

variables. This allows us to use the `llprover` [56] theorem prover, which at the time of writing does not support equality theories. The above formula is discharged in less than 20 ms.

2.11 Related work

Several papers develop type systems for (variants of) RCF [31–35, 57] but, with the exception of F* [35], they do not support resource-aware policies: in fact, even for simple linearity properties like injective agreement they rely on hand-written proofs [58].

F* [35] 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, however, 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 2.8 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 formulas 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. [59] 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 [57]. 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. [51] 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 [61–63]. However, 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 [31], 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 [35], 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 [64] have recently shown how to use behavioral contracts to link code written in an affine language to code written 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.

Various techniques have been proposed to statically analyze authenticity properties of cryptographic protocols [18, 19, 65–68], among which several types and effects systems [23,25–30,50,69,70]. These type systems incorporate ad-hoc mechanisms to deal with nonce handshakes and, thus, to enforce injective agreement properties. Our exponential serialization technique can be seen as a logic-based generalization of such mechanisms, independent of the language and the 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 AF7 type system does not allow to validate some specific nonce-handshake idioms, like the SOSH scheme [23]. Still, this can be recovered by extending AF7 with union and intersection types, as shown in [32, 33].

In previous work [1,2], we made initial steps towards the design of a sound system for resource-sensitive authorization, drawing on techniques from type systems for authentication and an affine extension of existing refinement type systems for the applied pi-calculus [71]. That work aims at analyzing cryptographic protocols as opposed to their implementations. Furthermore, such a type system is 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 chapter allows us to encode cryptography in the language using a standard sealing mechanism (cf. Section 2.7.2), 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, which in turn can be cleanly designed for the work presented in this chapter.

2.12 Conclusion

We presented AF7, the first type system for statically enforcing the (robust) safety of cryptographic protocol implementations with respect to authorization policies expressed in affine logic. AF7 benefits from the novel concept of exponential serialization to achieve a general and flexible treatment of affine formulas in distributed systems: we showed the effectiveness of this technique on two existing cryptographic protocols. We finally proposed AF7$_{\mathrm{alg}}$, an efficient, sound, and complete algorithmic variant of the type system, which is the key for a practical implementation of our analysis technique.

3
Type-Based Verification of Electronic Voting Protocols

E-voting protocols aim at achieving a wide range of sophisticated security properties and, consequently, commonly employ advanced cryptographic primitives. This makes their design as well as rigorous analysis quite challenging. As a matter of fact, existing automated analysis techniques, which are mostly based on automated theorem provers, are inadequate to deal with commonly used cryptographic primitives, such as homomorphic encryption and mix-nets, as well as some fundamental security properties, such as verifiability.

This chapter presents a novel approach based on refinement type systems for the automated analysis of e-voting protocols. Specifically, we design a generically applicable logical theory which, based on pre- and post-conditions for security-critical code, captures and guides the type-checker towards the verification of two fundamental properties of e-voting protocols, namely, vote privacy and verifiability. We further develop a code-based cryptographic abstraction of the cryptographic primitives commonly used in e-voting protocols, showing how to make the underlying algebraic properties accessible to automated verification through logical refinements. Finally, we demonstrate the effectiveness of our approach by developing the first automated analysis of Helios, a popular web-based e-voting protocol, using an off-the-shelf type-checker.

Publication. In this chapter we present the work that was presented under the title 'Type-Based Verification of Electronic Voting Protocols' at the 4th Conference on Principles of Security and Trust [5] in 2015. The corresponding technical report [6] was published in the IACR Cryptology ePrint Archive. Preliminary results on our analysis of privacy were joint work with Cyrille Wiedling and also included in his PhD thesis [72].

3.1 Introduction

While cryptographic protocols are notoriously difficult to design and their manual security analysis is extremely complicated in general, e-voting protocols are particularly tricky, since they aim at achieving sophisticated security properties, such as verifiability and coercion-resistance, and, consequently, employ advanced cryptographic primitives such as homomorphic encryptions, mix-nets, and zero-knowledge proofs. Not surprisingly, this makes the attack surface even larger, as witnessed by the number of attacks on e-voting protocols proposed in the literature (see e.g., [73–75]).

As we discussed in Chapter 1, the substantial research effort on the formal analysis of cryptographic protocols has led to the development of several automated tools based on symbolic abstractions of cryptography:

- Automated theorem provers build on a term-based abstraction of cryptography and proved successful in the enforcement of various trace properties [16–19] and even observational equivalence relations [20–22]. While some of these tools have also been used in the context of e-voting [76–79], they fall short of supporting the cryptographic primitives and security properties specific of this setting. For instance, none of them supports the commutativity property of homomorphic encryption that is commonly exploited to compute the final tally in a privacy-preserving manner (e.g., [80–82]), and the proof of complex properties like verifiability or coercion-resistance must be complemented by manual proofs [78, 83] or encodings [77] respectively, which are tedious and error-prone.

- Another line of research has focused on the design of *type systems* for cryptographic protocol analysis. Although they look promising, type systems have never been used in the context of e-voting protocols. This task is challenging since, for guiding the type-checking procedure, one needs to develop a dedicated logical theory, capturing the structure of e-voting systems and the associated security and privacy properties.

For further information about the related work we refer to Section 3.5.

Our contributions. We devise a novel approach based on refinement type systems for the formal verification of e-voting protocols. Specifically,

- we design a generically applicable logical theory based on pre- and post-conditions for security-critical code, which captures and guides the type-checker towards the verification of two fundamental properties, namely, vote privacy and verifiability;

- we formalize in particular three different verifiability properties (i.e., individual, universal, and end-to-end verifiability), proving for the first time that individual verifiability plus universal verifiability imply end-to-end verifiability, provided that ballots cannot be confused (no-clash property [75]);

- we develop a code-based cryptographic abstraction of the cryptographic primitives commonly used in e-voting protocols, including homomorphic encryption, showing how to make its commutativity and associativity properties accessible to automated verification through logical refinements;

- we demonstrate the effectiveness of our approach by analyzing Helios [82], a popular, state-of-the-art, voting protocol that has been used in several real-scale elections, including elections at Louvain-la-Neuve, Princeton, and among the IACR [84]. We analyze the two main versions of Helios that respectively use homomorphic encryption and mix-net based tally. For this we use F* [35], an off-the-shelf type-checker supporting the verification of trace properties and observational equivalence relations, as required for verifiability and vote privacy, through refinement and relational types, respectively. Analyzing Helios with homomorphic encryption was out of reach of existing tools due to the need of a theory that reflects the addition of the votes. A strength of our approach is that proof obligations involving such theories can be directly discharged to SMT solvers such as Z3 [85].

Outline. Section 3.2 reviews the most important concepts of type-based analysis and presents the Helios electronic voting protocol. Section 3.3 presents our modeling of individual, universal, and end-to-end verifiability and provides an analysis of the verifiability properties guaranteed by Helios. Section 3.4 explains our definition of vote privacy and shows how to enforce it on the example of Helios by using type-based analysis. Section 3.5 discusses related work. Section 3.6 concludes.

For the proof of Theorem 3.2, stating that no clash, individual, and universal verifiability entail end-to-end verifiability, we refer to the technical report [6]. This contribution is due to Véronique Cortier and Steve Kremer.

3.2 Background

We review the fundamental concepts underlying the typed-based analysis of security protocols and we present the Helios e-voting protocol that constitutes our case study.

3.2.1 Refinement types for cryptographic protocols

Review of RCF$_{\text{VOTE}}$. The protocols we analyze are implemented in a variant of Computational RCF [34] by Fournet et al., a λ-calculus with references, assumptions, and assertions. The only difference to the original version that was presented in [34] is the lack of the probabilistic sampling primitive, which is not required for our purposes. To distinguish this variant from the RCF variants used in Chapter 2 and Chapter 4 we will refer to it as RCF$_{\text{VOTE}}$. Below, we briefly review the syntax and semantics of the language. The full syntax of RCF$_{\text{VOTE}}$ is

a, b, c	label
x, y, z	variable
$h ::= \mathsf{inl} \mid \mathsf{inr} \mid \mathsf{fold}$	constructor
F, G	first-order formula
$M, N ::=$	value
x, y, z	variable
$()$	unit
(M, N)	pair
$h\ M$	construction
$\mathsf{fun}\ x \to A$	function (x bound in A)
read_a	reference read
write_a	reference write
$A, B ::=$	expression
M	value
$M\ N$	application
$\mathsf{let}\ x = A\ \mathsf{in}\ B$	let (x bound in B)
$\mathsf{let}\ (x, y) = M\ \mathsf{in}\ A$	split (x, y bound in A)
$\mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$	constructor match (x bound in A)
$\mathsf{ref}\ M$	reference creation
$\mathsf{assume}\ F$	assumption
$\mathsf{assert}\ F$	assertion

$\mathsf{true} \triangleq \mathsf{inl}\ ()$ and $\mathsf{false} \triangleq \mathsf{inr}\ ()$

Table 3.1: Syntax of RCF$_{\text{VOTE}}$ expressions

presented in Table 3.1. The semantics is standard: we refer to [34] for the complete formalization. Constructors, ranged over by h, include inl and inr, which are used to construct tagged unions, and fold, which is used to construct recursive data structures. Values, ranged over by M, N, comprise variables x, y, z, the unit value $()$, pairs (M, N), constructor applications $h\ M$, functions $\mathsf{fun}\ x \to A$, and functions read_a and write_a to read from and write to a memory location a, respectively. The syntax and semantics of expressions are mostly standard. $M\ N$ behaves as $A\{N/x\}$ (i.e., A where x is replaced by N) if $M = \mathsf{fun}\ x \to A$, otherwise it gets stuck; $\mathsf{let}\ x = A\ \mathsf{in}\ B$ evaluates A to M and then behaves as $B\{M/x\}$; $\mathsf{let}\ (x, y) = M\ \mathsf{in}\ A$ behaves as $A\{N/x, N'/y\}$ if $M = (N, N')$, otherwise it gets stuck; $\mathsf{match}\ M\ \mathsf{with}\ h\ x\ \mathsf{then}\ A\ \mathsf{else}\ B$ behaves as $A\{N/x\}$ if $M = h\ N$, as B otherwise; $\mathsf{ref}\ M$ allocates a fresh label a and returns the reading and writing functions $(\mathsf{read}_a, \mathsf{write}_a)$. The code is decorated with assumptions $\mathsf{assume}\ F$ and assertions $\mathsf{assert}\ F$. The former introduce logical formulas that are assumed to hold at a given program point, while the latter specify logical formulas that are expected to be entailed by the previously introduced assumptions.

Definition 3.1 (Safety)**.** *A closed expression A is safe iff the formulas asserted at run-time are logically entailed by the previously assumed formulas.*

The code is organized in *modules*, which are intuitively a sequence of function

$T, U, V ::=$		type
	unit	unit type
	α	type variable
	$\mu\alpha.T$	iso-recursive type (α bound in τ)
	$T + U$	sum type
	$x : T * U$	dependent pair type (x bound in U)
	$x : T \to U$	dependent function type (x bound in U)
	$x : T\{F\}$	dependent refinement type (x bound in F)

Table 3.2: Syntax of types (VOTE)

declarations. A module may export some of the functions defined therein, which can then be used by other modules: we let $B \cdot A$ denote the composition of modules B and A, where the functions exported by B may be used in A.

Types and typing judgements. Table 3.2 shows the syntax of types considered in this chapter. Types bool for boolean values and bytes for bitstrings can be constructed from unit by encoding[1]. The singleton unit type is populated by the value $()$; $\mu\alpha.T$ describes values of the form $\mathsf{fold}\ M$, where M has the unfolded type $T\{\mu\alpha.T/\alpha\}$; $T + U$ describes values of the form $\mathsf{inl}\ M$ or $\mathsf{inr}\ M$, where M has type T or U, respectively; the dependent type $x : T * U$ describes pairs of values (M, N), where M has type T and N has type $U\{M/x\}$; the dependent type $x : T \to U$ describes functions taking as input a value M of type T and returning a value of type $U\{M/x\}$; the dependent refinement type $x : T\{F\}$ describes values M of type T such that the logical formula $F\{M/x\}$ is entailed by the active assumptions. Notice that a refinement on the input of a function expresses a pre-condition, while a refinement on the output expresses a post-condition.

The typing judgement $I \vdash A : T$ says that expression A can be typed with type T in a typing environment I. Intuitively, a typing environment binds the free variables and labels in A to a type. The typing judgement $I \vdash B \rightsquigarrow I'$ says that under environment I module B is well-typed and exports the typed interface I'.

Modeling the protocol and the opponent. The protocol is encoded as a module, which exports functions defining the cryptographic library as well as the protocol parties. The latter are modeled as cascaded functions, which take as input the messages received from the network and return the pair composed of the value to be output on the network and the continuation code [2]. Concurrent communication is modeled by letting the opponent, which has access to the exported functions, act as a scheduler.

[1] E.g., boolean values are encoded as $\mathsf{true} \triangleq \mathsf{inl}\ ()$ and $\mathsf{false} \triangleq \mathsf{inr}\ ()$

[2] For the sake of readability we use the standard message-passing-style syntax in our examples and some additional syntactic sugar (e,g., sequential let declarations) that are easy to encode.

Modeling the cryptographic library. We rely on a sealing-based abstraction of cryptography [53, 54]. A seal for a type T consists of a pair of functions: the sealing function of type $T \to \mathsf{bytes}$ and the unsealing function of type $\mathsf{bytes} \to T$. The sealing mechanism is implemented by storing a list of pairs in a global reference that can only be accessed using the sealing and unsealing functions. The sealing function pairs the payload with a fresh, public value (the handle) representing its sealed version, and stores the pair in the list. The unsealing function looks up the public handle in the list and returns the associated payload. For symmetric cryptography, the sealing and unsealing functions are both private and naturally model encryption and decryption keys, respectively: a payload of type T is sealed to type bytes and can be sent over the untrusted network, while a message retrieved from the network with type bytes can be unsealed to its correct type T. Different cryptographic primitives, like public key encryptions and signature schemes, can be encoded in a similar way, by exporting the function modeling the public key to the opponent. We will give further insides on how to build sealing-based abstractions for more sophisticated cryptographic primitives, such as homomorphic encryptions and proofs of knowledge in Section 3.4.

Type-based verification. Assumptions and assertions can be used to express a variety of trace-based security properties. In this chapter, we consider policies expressed in first-order logic. For instance, consider the very simple e-voting protocol below, which allows everyone in possession of the signing key k_V, shared by all eligible voters, to cast arbitrarily many votes.

$$\begin{array}{ccc} V & & T \\ \mathsf{assume\ Cast}(v) & & \\ & \xrightarrow{\mathsf{sign}(k_V, v)} & \\ & & \mathsf{assert\ Count}(v) \end{array}$$

The assumption $\mathsf{Cast}(v)$ on the voter's side tracks the intention to cast vote v. The authorization policy $\forall v.\mathsf{Cast}(v) \Rightarrow \mathsf{Count}(v)$, which is further defined in the system as a global assumption expresses the fact that all votes cast by eligible voters should be counted. Since this is the only rule entailing $\mathsf{Count}(v)$, this rule actually captures a correspondence assertion: votes can be counted only if they come from eligible voters. The assertion $\mathsf{assert\ Count}(v)$ on the tallying authority's side expresses the expectation that vote v should be counted.

In order to type-check the code of authority T, it suffices to prove $\mathsf{Cast}(v)$ on the authority's side, which entails $\mathsf{Count}(v)$ through the authorization policy. Since the type-checking algorithm is modular (i.e., each party is independently analyzed) and $\mathsf{Cast}(v)$ is assumed on the voter's side, this formula needs to be conveyed to T. This is achieved by giving the vote v the *refinement type* $x : \mathsf{bytes}\{\mathsf{Cast}(x)\}$. In order to type v on the voter's side with such a type, v needs to be of type bytes and additionally, the formula $\mathsf{Cast}(v)$ needs to be entailed by the previous assumptions, which is indeed true in this case. In our sealing-based library for signatures signing corresponds to sealing a value and verification is modeled using the unsealing function and thus the types of signing and verification are $\mathsf{sigkey}(T) \triangleq T \rightarrow \mathsf{bytes}$ and $\mathsf{verkey}(T) \triangleq \mathsf{bytes} \rightarrow T$, while the types of the signing and verification functions are $\mathsf{sig} : \mathsf{sigkey}(T) \rightarrow T \rightarrow \mathsf{bytes}$ and $\mathsf{ver} : \mathsf{verkey}(T) \rightarrow \mathsf{bytes} \rightarrow T$, respectively.[3] Here T is $x : \mathsf{bytes}\{\mathsf{Cast}(x)\}$, thereby imposing a pre-condition on the signing function (before signing x, one has to assume the formula $\mathsf{Cast}(x)$) and a post-condition on the verification function (after a successful verification, the formula $\mathsf{Cast}(x)$ is guaranteed to hold for the signed x).

When reasoning about the implementations of cryptographic protocols, we are interested in the safety of the protocol against an arbitrary opponent.

Definition 3.2 (Opponent and Robust Safety)**.** *A closed expression O is an opponent iff O contains no assumptions or assertions. A closed module A is robustly safe w.r.t. interface I iff for all opponents O such that $I \vdash O : T$ for some type T, $A \cdot O$ is safe.*

Following the approach advocated in [34], the typed interface I exported to the opponent is supposed to build exclusively on the type bytes, without any refinement. This means that the attacker is not restricted by any means and can do whatever it wants with the messages received from the network, except for performing invalid operations that would lead it to be stuck (e.g., treating a pair as a function). In fact, the well-typedness assumption for the opponent just makes sure that the only free variables occurring therein are the ones exported by the protocol module. Robust safety can be statically enforced by type-checking, as stated below.

Theorem 3.1 (Robust Safety)**.** *If $\emptyset \vdash A \rightsquigarrow I$ then A is robustly safe w.r.t. I.*

3.2.2 Helios

Helios [82] is a verifiable and privacy-preserving e-voting system. It has been used in several real-life elections such that student elections at the University of Louvain-la-Neuve or at Princeton. It is now used by the IACR to elect its board since 2011 [84]. The current implementation of Helios (Helios 2.0) is based on homomorphic encryption, which makes it possible to decrypt only the aggregation of the ballots as opposed to the individual ballots. Homomorphic tally, however, requires encrypted ballots to be split in several ciphertexts, depending on the number of candidates. For example, in case of 4 candidates and a vote for the second one, the encrypted ballot would be $\{0\}^{r_1}_{\mathsf{pk}}, \{1\}^{r_2}_{\mathsf{pk}}, \{0\}^{r_3}_{\mathsf{pk}}, \{0\}^{r_4}_{\mathsf{pk}}$. In case the number of candidates is high, the size of a ballot and the computation time become large. Therefore, there exists a variant of Helios that supports mix-net-based tally: ballots are shuffled and re-randomized before being decrypted. Both variants co-exist since they both offer advantages: mix-nets can cope with

[3] We note that the verification function only takes the signature as an input, checks whether it is indeed a valid signature and if so, retrieves the corresponding message that was signed. This is a standard abstraction and used for convenience, an alternate approach would be to have verification take both the signature and message as an input and return a boolean value. The sealing-based library functions for both versions are very similar.

a large voting space while homomorphic tally eases the decryption phase (only one ballot needs to be decrypted, no need of mixers). We present here both variants of Helios, which constitute our case studies. For simplicity, in the case of homomorphic tally, we assume that voters are voting either 0 or 1 (referendum).

The voting process in Helios is divided in two main phases. The bulletin board is a public webpage that starts initially empty. Votes are encrypted using a public key pk. The corresponding decryption key dk is shared among trustees. For privacy, the trust assumption is that at least one trustee is honest (or that the trustees do not collaborate).

Voting phase. During the voting phase, each voter encrypts her vote v using the public key pk of the election.She then sends her encrypted vote $\{v\}^r_{\mathsf{pk}}$ (where r denotes the randomness used for encrypting), together with some auxiliary data aux, to the bulletin board through an authenticated channel. In the homomorphic version of Helios, aux contains a zero-knowledge proof that the vote is valid, that is 0 or 1. This avoids that a voter gives e.g. 100 votes to a candidate. In the mix-net variant of Helios, aux is empty. Provided that the voter is entitled to vote, the bulletin board adds the ballot $\{v\}^r_{\mathsf{pk}}, \mathsf{aux}$ to the public list. The voter should check that her ballot indeed appears on the public bulletin board.

The voter's behavior is described in Figure 3.1. It corresponds to the mix-net version but could be easily adapted to the homomorphic version. Note that this description contains assume and assert annotations that intuitively represent different states of the voter's process. These annotations are crucially used to state verifiability, cf. Section 3.3.

The voting phase also includes an optional audit phase allowing the voter to audit her ballot instead of casting it. In that case, her ballot and the corresponding randomness are sent to a third party that checks whether the correct choice has been encrypted. Here, we do not model the auditing phase, since a precise characterization would probably require probabilistic reasoning, which goes beyond the scope of this work.

Tallying phase. Once the voting phase is over, the bulletin board contains a list of ballots $\{v_1\}^{r_1}_{\mathsf{pk}}, \dots, \{v_n\}^{r_n}_{\mathsf{pk}}$ (we omit the auxiliary data). We distinguish the two variants.

- *Homomorphic tally.* The ballots on the bulletin board are first homomorphically combined. Since $\{v\}^r_{\mathsf{pk}} * \{v'\}^{r'}_{\mathsf{pk}} = \{v + v'\}^{r+r'}_{\mathsf{pk}}$ anyone can compute the encrypted sum of the votes $\{\sum_{i=1}^n v_i\}^{r*}_{\mathsf{pk}}$. Then the trustees collaborate to decrypt this ciphertext. Their computation yields $\sum_{i=1}^n v_i$ and a proof of correct decryption.

- *Mix-net tally.* Ballots are shuffled and re-randomized, yielding

  $$\{v_{i_1}\}^{r'_1}_{\mathsf{pk}}, \dots, \{v_{i_n}\}^{r'_n}_{\mathsf{pk}}$$

  with a proof of correct permutation. This mixing is performed successively by several mixers. For privacy, the trust assumption is that as least one

```
Voter($id, v$) =  assume Vote($id, v$);
                  let $r$ = new() in
                  let $b$ = enc(pk, $v, r$) in
                  assume MyBallot($id, v, b$);
                  send(net, $b$);
                  let $bb$ = recv(net) in
                  if $b \in bb$ then
                  assert VHappy($id, v, bb$)
```

Figure 3.1: Modeling of a voter.

mix-net is honest (that is, will not leak the permutation). Then the trustees collaborate to decrypt each (re-randomize) ciphertext and provide a corresponding proof of correct decryption.

3.3 Verifiability

Verifiability is a key property in both electronic as well as paper-based voting systems. Intuitively, verifiability ensures that the announced result corresponds to the votes such as intended by the voters. Verifiability is typically split into several sub-properties.

- *Individual verifiability* ensures that a voter is able to check that her ballot is on the bulletin board.
- *Universal verifiability* ensures that any observer can verify that the announced result corresponds to the (valid) ballots published on the bulletin board.

Symbolic models provide a precise definition of these notions [86].

The overall goal of these two notions is to guarantee *end-to-end verifiability*: if a voter correctly follows the election process her vote is counted in the final result. In our terminology, *strong end-to-end verifiability* additionally guarantees that at most k dishonest votes have been counted, where k is the number of compromised voters. This notion of strong end-to-end verifiability includes the notion of what is called *eligibility verifiability* in [86]. For simplicity, we focus here on end-to-end verifiability.

We will now explain our modeling of individual, universal, and end-to-end verifiability. One of our contributions is a logical formalization of these properties that enables the use of off-the-shelf verification techniques, in our case a type system, at least in the case of individual and universal verifiability. End-to-end verifiability may be more difficult to type-check directly. Instead, we formally prove for the first time that individual and universal verifiability entail end-to-end verifiability provided that there are no "clash attacks" [75]. A clash attack typically arises when two voters are both convinced that the same ballot b is "their" own ballot. In that case, only one vote will be counted instead of two. The fact that individual and universal verifiability entail end-to-end verifiability has two main advantages. First, it provides a convenient proof technique: it is sufficient to prove individual and universal verifiability, which as we will show can be done with the help of a type-checker. Second, our results provide a better understanding of the relation between the different notions of verifiability.

Notations. Before presenting our formal model of verifiability we introduce a few notations. Voting protocols aim at counting the votes. Formally, a *counting function* is a function $\rho : \mathbb{V}^* \to R$, where $\mathbb{V}$ is the vote space and R the result space. A typical voting function is the number of votes received by each candidate. By a slight abuse of notation, we may consider $\rho(l)$ where l is a list of votes instead of a sequence of votes.

If l is a list, $\#l$ denotes the size of l and $l[i]$ refers to the ith element of the list. $a \in l$ holds if a is an element of l. Given $a_1, \ldots, a_n$, we denote by $\{\!|a_1, \ldots, a_n|\!\}$ the corresponding multiset. $\subseteq_m$ denotes multiset inclusion. Assume l_1, l_2 are lists; by a slight abuse of notation, we may write $l_1 \subseteq_m l_2$ where l_1, l_2 are viewed as multisets. We also write $l_1 =_m l_2$ if the two lists have the same multisets of elements.

In order to express verifiability and enforce it using a type system, we rely on the following assumptions:

- $\mathsf{assume\ Vote}(id, v, c)$ means that voter id intends to vote for c possibly using some credential c. This predicate should hold as soon as the voter starts to vote: he knows for whom he is going to vote.

- $\mathsf{assume\ MyBallot}(id, v, b)$ means that voter id thinks that ballot b contains her vote v. In case votes are sent in clear, b is simply the vote v itself. In the case of Helios, we have $b = \{v\}^r_{\mathsf{pk}}, \mathsf{aux}$. Typically, this predicate should hold as soon as the voter (or her computer) has computed the ballot.

An example of where and how to place these predicates for Helios can be found in Figure 3.1. The credential c is omitted since there is no use of credentials in Helios.

3.3.1 Individual verifiability

Intuitively, individual verifiability enforces that whenever a voter completes her process successfully, her ballot is indeed in the ballot box. Formally we define the predicate VHappy as follows:

$$\mathsf{assume\ VHappy}(id, v, c, bb) \;\Leftrightarrow\; \mathsf{Vote}(id, v, c) \;\wedge\; \exists b \in bb.\ \mathsf{MyBallot}(id, v, b)$$

This predicate should hold whenever voter id has finished her voting process, and believes she has voted for v. At that point, it should be the case that the ballot box bb contains the vote v (in some ballot). We therefore annotate the voter function with the assertion $\mathsf{assert\ VHappy}(id, v, c, bb)$. This annotation is generally the final instruction, see Figure 3.1 for the Helios example.

Definition 3.3 (Individual Verifiability)**.** *A protocol with security annotations*

```
$\mathsf{Judge}(bb, r) =$  let $vbb = \mathsf{recv}(\mathsf{net})$ in
                   let $zkp = \mathsf{recv}(\mathsf{net})$ in
                   if $vbb = \mathsf{removeDuplicates}(bb) \wedge \mathsf{check\_zkp}(zkp, vbb, r)$ then
                   assert $\mathsf{JHappy}(bb, r)$
```

Figure 3.2: Judge function for Helios

- assume $\mathsf{Vote}(id, v, c)$*,* assume $\mathsf{MyBallot}(id, v, b)$*;*
- *and* assert $\mathsf{VHappy}(id, v, c, bb)$

as described above guarantees individual verifiability *if it is robustly safe.*

3.3.2 Universal verifiability

Intuitively, universal verifiability guarantees that anyone can check that the result corresponds to the ballots present in the ballot box. Formally, we assume a program $\mathsf{Judge}(bb, r)$ that checks whether the result r is valid w.r.t. ballot box bb. Typically, Judge does not use any secret and could therefore be executed by anyone. We simply suppose that Judge contains assert $\mathsf{JHappy}(bb, r)$ at some point, typically when all the verification checks succeed. For Helios, the Judge program is displayed Figure 3.2. We first introduce a few additional predicates that we use to define the predicate JHappy.

Good sanitization. Once the voting phase is closed, the tallying phase proceeds in two main phases. First, some "cleaning" operation is performed in bb, e.g., invalid ballots (if any) are removed and duplicates are weeded, resulting in the sanitized valid bulletin board vbb. Intuitively, a good cleaning function should not remove ballots that correspond to honest votes. We therefore define the predicate $\mathsf{GoodSan}(bb, vbb)$ to hold if the honest ballots of bb are not removed from vbb.

$$\text{assume } \mathsf{GoodSan}(bb, vbb) \Leftrightarrow \forall b.[(b \in bb \ \wedge \ \exists id, v.\mathsf{MyBallot}(id, v, b)) \Rightarrow b \in vbb]$$

Good counting. Once the ballot box has been sanitized, ballots are ready to be tallied. A good tallying function should count the votes "contained" in the ballots. To formally define that a vote is "contained" in a ballot, we consider a predicate $\mathsf{Wrap}(v, b)$ that is left undefined, but has to satisfy the following properties:

- any well-formed ballot b corresponding to some vote v satisfies: $\mathsf{MyBallot}(id, v, b) \ \Rightarrow \ \mathsf{Wrap}(v, b)$
- a ballot cannot wrap two distinct votes: $\mathsf{Wrap}(v_1, b) \wedge \mathsf{Wrap}(v_2, b) \ \Rightarrow \ v_1 = v_2$

If these two properties are satisfied, we say that Wrap is *voting-compliant.* For a given protocol, the definition Wrap typically follows from the protocol specification.

Example 3.1. *In the Helios protocol, the* Wrap *predicate is defined as follows.*

$$\mathsf{assume}\ \mathsf{Wrap}(v,b)\ \Leftrightarrow\ \exists r.\ \mathsf{Enc}(v,r,\mathsf{pk},b)$$

where $\mathsf{Enc}(v,r,\mathsf{pk},b)$ *is a predicate that holds if* b *is the result of the encryption function called with parameters* pk*,* v *and* r*. It is easy to see that* Wrap *is voting-compliant and this can in fact be proved using a type-checker. It is sufficient to add the annotations*

- $\mathsf{assert}\ \mathsf{MyBallot}(id,v,b) \Rightarrow \mathsf{Wrap}(v,b)$ *and*
- $\mathsf{assert}\ \forall v_1,v_2.\ \mathsf{Wrap}(v_1,b) \wedge \mathsf{Wrap}(v_2,b) \Rightarrow v_1 = v_2$

to the voter function (Figure 3.1) just after the MyBallot *assumption. The second assertion is actually a direct consequence of our modeling of encryption which implies that a ciphertext cannot decrypt to two different plaintexts.*

We are now ready to state when votes have been correctly counted: the result should correspond to the counting function ρ applied to the votes contained in each ballot. Formally, we define $\mathsf{GoodCount}(vbb,r)$ to hold if the result r corresponds to counting the votes of *rlist*, i.e., the list of votes obtained from the ballots in vbb'. The list vbb' is introduced for technical convenience and either denotes the list of valid votes vbb itself (in the homomorphic variant) or any arbitrary permutation of vbb (for mix-nets).

$$\begin{array}{ll}\mathsf{assume}\ \mathsf{GoodCount}(vbb,r)\ \Leftrightarrow\ \exists vbb',rlist.\,[& \#vbb = \#rlist\ \wedge\ vbb =_m vbb' \wedge \\ & \forall b,i.[vbb'[i] = b \\ & \Rightarrow\ \exists v.(\mathsf{Wrap}(v,b) \wedge (rlist[i] = v))] \wedge \\ & r = \rho(rlist)\ ]\end{array}$$

Note that the definition of GoodCount is parameterized by the counting function ρ of the protocol under consideration. We emphasize that for $\mathsf{GoodCount}(vbb,r)$ to hold, the sanitized bulletin board may only contain correctly wrapped ballots, i.e., we assume that the sanitization procedure is able to discard invalid ballots. In the case of mix-net-based Helios we therefore require that the sanitization discards any ballots that do not decrypt. This can for instance be achieved by requiring a zero knowledge proof that the submitted bitstring is a correct ciphertext. We may however allow that a ballot decrypts to an invalid vote, as such votes can be discarded by the tallying function.

Universal verifiability. Finally, universal verifiability enforces that whenever the verification checks succeed (that is, the Judge's program reaches the JHappy assertion), then GoodSan and GoodCount should be guaranteed. Formally, we define the predicate

$$\mathsf{assume}\ \mathsf{JHappy}(bb,r)\ \Leftrightarrow\ \exists vbb.\ (\mathsf{GoodSan}(bb,vbb)\ \wedge\ \mathsf{GoodCount}(vbb,r))$$

and add the annotation $\mathsf{assert}\ \mathsf{JHappy}(bb,r)$ at the end of the judge function.

Definition 3.4 (Universal Verifiability). *A protocol with security annotations*

- assume $\mathsf{MyBallot}(id, v, b)$, *and*
- assert $\mathsf{JHappy}(bb, r)$

as described above guarantees universal verifiability *if it is robustly safe and the predicate* $\mathsf{Wrap}(v, b)$ *is voting-compliant.*

3.3.3 End-to-end verifiability

End-to-end verifiability is somehow simpler to express. It ensures that whenever the result is valid (that is, the judge has reached his final state), the result contains at least all the votes of the voters that have reached their final states. In other words, voters that followed the procedure are guaranteed that their vote is counted in the final result. To formalize this idea we define the predicate EndToEnd as follows:

$$\begin{array}{l} \mathsf{assume}\ \ \mathsf{EndToEnd} \Leftrightarrow \forall bb, r, id_1, \ldots, id_n, v_1, \ldots, v_n, c_1, \ldots, c_n. \\ \qquad\qquad (\mathsf{JHappy}(bb, r) \wedge \mathsf{VHappy}(id_1, v_1, c_1, bb) \wedge \ldots \wedge \mathsf{VHappy}(id_n, v_n, c_n, bb)) \\ \qquad\qquad\quad \Rightarrow \exists rlist.\ r = \rho(rlist) \wedge \{\!|v_1, \ldots, v_n|\!\} \subseteq_m rlist \end{array}$$

To ensure that this predicate holds we can again add a final assertion assert EndToEnd.

Definition 3.5 (End-to-End Verifiability). *A protocol with security annotations*

- assume $\mathsf{Vote}(id, v, c)$, assume $\mathsf{MyBallot}(id, v, b)$;
- *and* assert $\mathsf{VHappy}(id, v, c, bb)$, assert $\mathsf{JHappy}(bb, r)$, assert EndToEnd

as described above guarantees end-to-end verifiability *if it is robustly safe.*

For simplicity, we have stated end-to-end verifiability referring explicitly to a bulletin board. It is however easy to state our definition more generally by letting bb be any form of state of the protocol. This more general definition does not assume a particular structure of the protocol, as it is also the case in a previous definitions of end-to-end verifiability in the literature [87].

It can be difficult to directly prove end-to-end verifiability using a type-checker. An alternative solution is to show that it is a consequence of individual and universal verifiability. However, it turns out that individual and universal verifiability are actually not sufficient to ensure end-to-end verifiability. Indeed, assume that two voters id_1 and id_2 are voting for the same candidate v. Assume moreover that they have built the same ballot b. In case of Helios, this could be the case if voters are using a bad randomness generator. Then a malicious bulletin board could notice that the two ballots are identical and could display only one of the two. The two voters would still be "happy" (they can see their ballot on the bulletin board) as well as the judge since the tally would correspond to the bulletin board. However, only one vote for v would be counted instead of two. Such a scenario has been called a clash attack [75].

We capture this property by the predicate NoClash defined as follows.

$$\mathsf{NoClash} \Leftrightarrow \forall id_1, id_2, v_1, v_2, b.\ \ \mathsf{MyBallot}(id_1, v_1, b)\ \wedge\ \mathsf{MyBallot}(id_2, v_2, b) \\ \Rightarrow\ id_1 = id_2\ \wedge\ v_1 = v_2$$

The assertion assert NoClash is then added after the assumption MyBallot.

Definition 3.6 (No Clash)**.** *A protocol with security annotations*

- assume MyBallot(id, v, b) *and*
- assert NoClash

as described above guarantees no clash *if it is robustly safe.*

We can now state our result that no clash, individual, and universal verifiability entail end-to-end verifiability. The proof is due to Véronique Cortier and Steve Kremer and can be found in the technical report [6] of the corresponding conference paper that this chapter builds on.

Theorem 3.2. *If a protocol guarantees individual and universal verifiability as well as no clash, then it satisfies end-to-end verifiability.*

3.3.4 Verifiability analysis of Helios

Using the F* type-checker (version 0.7.1-alpha) we have analyzed both the mix-net and homomorphic versions of Helios. The corresponding files can be found in [88]. The (simplified) model of the voter and judge functions is displayed in Figures 3.1 and 3.2.

Helios with mix-nets. Using F*, we automatically proved both individual and universal verifiability. As usual, we had to manually define the types of the functions, which crucially rely on refinement types to express the expected pre- and post-conditions. For example, for universal verifiability, one has to show that GoodSan and GoodCount hold whenever the judge agrees with the tally. For sanitization, the judge verifies that vbb is a sublist of bb, where duplicate ballots have been removed. Thus, the type-checker can check that the function removeDuplicates(bb) returns a list vbb whose type is a refinement stating that $x \in bb \Rightarrow x \in vbb$, which allows us to prove GoodSan. Regarding GoodCount, the judge verifies a zero-knowledge proof that ensures that any vote in the final result corresponds to an encryption on the sanitized bulletin board. Looking at the type of the check_zkp function we see that this information is again conveyed through a refinement of the boolean type returned by the function:

$$\mathsf{check\_zkp} : zkp : \mathsf{bytes} \rightarrow vbb : \mathsf{list\ ballot} \rightarrow res : \mathsf{result} \rightarrow b : \mathsf{bool}\{b = \mathsf{true} \Rightarrow \varphi\}$$

$$\text{where } \varphi \triangleq \exists vbb'.\,[\ \ \#vbb = \#res\ \wedge\ vbb =_m vbb'\ \wedge \\ \forall b, i.[vbb'[i] = b \Rightarrow\ \exists v, r.(\mathsf{Enc}(v, r, \mathsf{pk}, b)\ \wedge (res[i] = v))]\ \ ]$$

In the case where check_zkp returns true we have that the formula φ holds. The formula φ is similar to the GoodCount predicate (with ρ being the identity function for mix-net based Helios) except that it ensures that a ballot is an encryption, rather than a wrap. This indeed reflects that the zero-knowledge proof used in the protocol provides exactly the necessary information to the judge to conclude that the counting was done correctly.

The *no clash* property straightforwardly follows from observing that the logical predicate $\mathsf{MyBallot}(id, v, b)$ is assumed only once in the voter's code, that each voter has a distinct id, and that, as argued for $\mathsf{Wrap}(v, b)$, the same ciphertext cannot decrypt to two different plaintexts. By Theorem 3.2, we can conclude that the mix-net version of Helios indeed satisfies end-to-end verifiability.

Type-checkers typically support lists with the respective functions (length, membership test, etc.). As a consequence, we prove individual and universal verifiability *for an arbitrary number of dishonest voters*, while only a fixed number of dishonest voters can typically be considered with other existing protocol verification tools.

Helios with homomorphic tally. The main difference with the mix-net version is that each ballot additionally contains a zero-knowledge proof, that ensures that the ballot is an encryption of either 0 or 1. The judge function also differs in the tests it performs. In particular, to check that the counting was performed correctly, the judge verifies a zero-knowledge proof that ensures that the result is the sum of the encrypted votes that are on the sanitized bulletin board. This ensures in turn that the result corresponds to the sum of the votes. Considering the "sum of the votes" is out of reach of classical automated protocol verification tools. Here, F* simply discharges the proof obligations involving the integer addition to the Z3 solver [85] which is used as a back-end.

Finally, as for the mix-net based version, we proved individual and universal verifiability using F*, while the *no clash* property relies on (the same) manual reasoning. Again, we conclude that end-to-end verifiability is satisfied using Theorem 3.2.

3.4 Privacy

The secrecy of a ballot is of vital importance to ensure that the political views of a voter are not known to anyone. Vote privacy is thus considered a fundamental and universal right in modern democracies.

In this section we review the definition of vote privacy based on observational equivalence [83] and present a type-based analysis technique to verify this property using RF*, an off-the-shelf type-checker. We demonstrate the usefulness of our approach by analyzing vote privacy in the homomorphic variant of Helios, which was considered so far out of the scope of automated verification techniques.

3.4.1 Definition of privacy

Observational equivalence. We first introduce the concept of observational equivalence, a central tool to capture indistinguishability properties. The idea is that two runs of the same program with different secrets should be indistinguishable for any opponent. The definition is similar to the natural adaption of the one presented in [34] to a deterministic, as opposed to probabilistic, setting.

Definition 3.7 (Observational Equivalence)**.** *For all modules A, B we say that A and B are observationally equivalent, written $A \approx B$, iff they both export the same interface I and and for all opponents O that are well-typed w.r.t the interface I it holds that $A \cdot O \to^* M$ iff $B \cdot O \to^* M$ for all closed values M.*

Here, $A \to^* N$ denotes that expression A eventually evaluates to value N, according to the semantic reduction relation.

Privacy. We adopt the definition of vote privacy presented in [76]. This property ensures that the link between a voter and her vote is kept secret. Intuitively, in the case of a referendum this can only be achieved if at least two honest voters exist, since otherwise all dishonest voters could determine the single honest voter's vote from the final tally by colluding. Furthermore, both voters must vote for different parties, thus counter-balancing each other's vote and ensuring that it is not known who voted for whom. Our definition of privacy thus assumes the existence of two honest voters Alice and Bob and two candidates v_1 and v_2. We say that a voting system guarantees privacy if a protocol run in which Alice votes v_1 and Bob votes v_2 is indistinguishable (i.e., observationally equivalent) from the protocol run in which Alice votes v_2 and Bob votes v_1.

In the following, we assume the voting protocol to be defined as $\mathsf{fun}\ (v_A, v_B) \to \mathsf{S}[\mathsf{Alice}(v_A), \mathsf{Bob}(v_B)]$. The two honest voters Alice and Bob are parameterized over their votes v_A and v_B. Here, $\mathsf{S}[\bullet, \bullet]$ describes a two-hole context (i.e., an expression with two holes), which models the behavior of the cryptographic library, the public bulletin board, and the election authorities (i.e., the surrounding system).

Definition 3.8 (Vote Privacy)**.** $P = \mathsf{fun}\ (v_A, v_B) \to \mathsf{S}[\mathsf{Alice}(v_A), \mathsf{Bob}(v_B)]$ *guarantees vote privacy iff for any two votes v_1, v_2 it holds that $P(v_1, v_2) \approx P(v_2, v_1)$.*

3.4.2 RF*: A type system for observational equivalence properties

To prove privacy for voting protocols we rely on RF*, an off-the-shelf existing type-checker that can be used to enforce indistinguishability properties. RF* was introduced by Barthe et al. [89] and constitutes the relational extension of the F* type-checker [35]. The core idea is to let refinements reason about two runs (as opposed to a single one) of a protocol. Such refinements are called *relational refinements.* A relational refinement type has the form $x : T\{|F|\}$, where the formula F may capture the instantiation of x in the left run of the expression that is to be type-checked, denoted $\mathsf{L}\ x$, as well as the instantiation of x in

```
Alice $v_A$ =
let $b_A = \text{create\_ballot}_A(v_A)$ in
send$(c_A, b_A)$
```

Figure 3.3: Model of Alice

```
Bob $v_B$ =
let $b_B = \text{create\_ballot}_B(v_B)$ in
send$(c_B, b_B)$
```

Figure 3.4: Model of Bob

the right run, denoted $\mathsf{R}\ x$. Formally, $A : x : T\{|F|\}$ means that whenever A evaluates to M_L and M_R in two contexts that provide well-typed substitutions for the free variables in A, then the formula $F\{^{M_L}/_{\mathsf{L}\ x}\}\{^{M_R}/_{\mathsf{R}\ x}\}$ is valid. We note that relational refinements are strictly more expressive than standard refinements. For instance, $x : \mathsf{bytes}\{H(x)\}$ can be encoded as $x : \mathsf{bytes}\{|H(\mathsf{L}\ x) \wedge H(\mathsf{R}\ x)|\}$. A special instance of relational refinement types is the so-called *eq-type*. Eq-types specify that a variable is instantiated to the same value in both the left and the right protocol run. Formally, $\mathsf{eq}\ T \triangleq x : T\{|\mathsf{L}\ x = \mathsf{R}\ x|\}$. The authors show how such types can be effectively used to verify both non-interference and indistinguishability properties.

3.4.3 Type-based verification of vote privacy

In the following, we show how to leverage the aforementioned technique to statically enforce observational equivalence and, in particular, vote privacy. The key observation is that whenever a value M is of type eq bytes it can safely be published, i.e., given to the opponent. Intuitively, this is the case since in both protocol runs, this value will be the same, i.e., the opponent will not be able to observe any difference. Given that both runs consider the same opponent O, every value produced by the opponent must thus also be the same in both runs, which means it can be typed with eq bytes.

We denote typed interfaces that solely build on eq bytes by I_{eq} and following the above intuition state that if a voting protocol can be typed with such an interface, the two runs where (i) Alice votes v_1, Bob votes v_2 and (ii) Alice votes v_2, Bob votes v_1 are observationally equivalent, since no opponent will be able to distinguish them.

Theorem 3.3 (Privacy by Typing). *For all $P = \mathsf{fun}\ (v_A, v_B) \rightarrow \mathsf{S}[\mathsf{Alice}(v_A), \mathsf{Bob}(v_B)]$ and all M, M', v_1, v_2 such that $M : x : \mathsf{bytes}\{|\mathsf{L}\ x = v_1 \wedge \mathsf{R}\ x = v_2|\}$ and $M' : x : \mathsf{bytes}\{|\mathsf{L}\ x = v_2 \wedge \mathsf{R}\ x = v_1|\}$ it holds that if $\emptyset \vdash P(M, M') \rightsquigarrow I_{\mathsf{eq}}$, then P provides vote privacy.*

Modeling a protocol for privacy verification. We demonstrate our approach on the example of Helios with homomorphic encryption. For simplicity, we consider one ballot box that owns the decryption key dk and does the complete tabulation. An informal description of Alice and Bob's behavior is displayed in Figures 3.3 and 3.4, respectively. The voters produce the relationally refined ballots using the ballot creation functions create_ballot$_A$, create_ballot$_B$ respectively.

```
BB = let $b_A = \mathsf{recv}(c_A)$ in
     let $b_B = \mathsf{recv}(c_B)$ in
     send(net, $(b_A, b_B)$);
     let $b_O = \mathsf{recv}(\mathsf{net})$ in
     if check_zkp($b_A$) true then
     match check_zkp($b_B$) with true then
     match check_zkp($b_O$) with true then
     match $(b_A \neq b_O \wedge b_A \neq b_B \wedge b_B \neq b_O)$ with true then
     let $b_{AB}$ = add_ballot($b_A, b_B$) in
     let $b_{ABO}$ = add_ballot($b_{AB}, b_O$) in
     let $result$ = dec_ballot($b_{ABO}$) in
     send(net, $result$)
```

Figure 3.5: Model of the ballot box

The ballots b_A, b_B consist of the randomized homomorphic encryption of the votes and a zero-knowledge proof of correctness and knowledge of the encrypted vote. The ballots are then sent to the ballot box over secure https-connections c_A and c_B respectively.

The behavior of the ballot box is described in Figure 3.5. For the sake of simplicity, we consider the case of three voters. The ballot box receives the ballots of Alice and Bob and publishes them on the bulletin board. It then receives the ballot of the opponent and checks that the proofs of validity of all received ballots succeed. Furthermore, it checks that all ballots are distinct before performing homomorphic addition on the ciphertexts. The sum of the ciphertexts is then decrypted and published on the bulletin board.

Intuitively, all outputs on the network are of type eq bytes, since (i) all ballots are the result of an encryption that keeps the payload secret and thus gives the opponent no distinguishing capabilities, and (ii) the homomorphic sum of all ciphertexts $b_{ABO} = \{v_A + v_B + v_O\}_{\mathsf{pk}}$ is the same in both runs of the protocol up to commutativity. Indeed, $\mathsf{L}\ b_{ABO} = \{v_1 + v_2 + v_O\}_{\mathsf{pk}}$ and $\mathsf{R}\ b_{ABO} = \{v_2 + v_1 + v_O\}_{\mathsf{pk}} = \mathsf{L}\ b_{ABO}$.

However, since the application of the commutativity rule happens on the level of plaintexts, while the homomorphic addition is done one level higher-up on ciphertexts, we need to guide the type-checker in the verification process.

Sealing-based library for voting. While privacy is per se not defined by logical predicates, we rely on some assumptions to describe properties of the cryptographic library, such as homomorphism and validity of payloads, in order to guide the type-checker in the derivation of eq-types. The (simplified) type of the sealing reference for homomorphic encryption with proofs of validity is given below:[4]

[4] The actual library includes *marshaling* operations, which we omit for simplicity.

$$\begin{aligned}&m : \mathsf{bytes} * c : \mathsf{eq\ bytes}\{|\mathsf{Enc}(m,c) \wedge \mathsf{Valid}(c)\wedge\\&(\mathsf{FromA}(c) \vee \mathsf{FromB}(c) \vee (\mathsf{FromO}(c) \wedge \mathsf{L}\ m = \mathsf{R}\ m))|\}\end{aligned}$$

Here, predicates FromA, FromB, FromO are used to specify whether an encryption was done by Alice, Bob or the opponent, while $\mathsf{Enc}(m, c)$ states that c is the ciphertext resulting from encrypting m and $\mathsf{Valid}(c)$ reflects the fact that the message corresponds to a valid vote, i.e., a validity proof for c can be constructed. Note that if a ballot was constructed by the opponent, the message stored therein must be the same in both runs ($\mathsf{L}\ m = \mathsf{R}\ m$), i.e., the message must have been of type eq bytes. These logical predicates are assumed in the sealing functions used by Alice, Bob, and the opponent, respectively. These functions, used to encode the public key, share the same code, and in particular they access the same reference, and only differ in the internal assumptions.

Similarly, there exist three ballot creation functions $\mathsf{create\_ballot}_A$, $\mathsf{create\_ballot}_B$, and $\mathsf{create\_ballot}_O$, used by Alice, Bob and the opponent, respectively, only differing in their refinements and internal assumptions. Their interfaces are listed below:

$$\begin{aligned}\mathsf{create\_ballot}_A :\ & m : x : \mathsf{bytes}\{|\mathsf{L}\ x = v_1 \wedge \mathsf{R}\ x = v_2|\} \rightarrow\\& c : \mathsf{eq\ bytes}\{|\mathsf{Enc}(m,c) \wedge \mathsf{FromA}(c)|\}\\\mathsf{create\_ballot}_B :\ & m : x : \mathsf{bytes}\{|\mathsf{L}\ x = v_2 \wedge \mathsf{R}\ x = v_1|\} \rightarrow\\& c : \mathsf{eq\ bytes}\{|\mathsf{Enc}(m,c) \wedge \mathsf{FromB}(c)|\}\\\mathsf{create\_ballot}_O :\ & \mathsf{eq\ bytes} \rightarrow \mathsf{eq\ bytes}\end{aligned}$$

Notice that, as originally proposed in [89], the result of probabilistic encryption (i.e., the ballot creation function) is given an eq bytes type, reflecting the intuition that there always exist two randomnesses, which are picked with equal probability, that make the ciphertexts obtained by encrypting two different plaintexts identical, i.e., probabilistic encryption does not leak any information about the plaintext.

The interfaces for the functions dec_ballot, check_zkp, add_ballot for decryption, validity checking of the proofs, and homomorphic addition are listed below. The public interfaces for the latter two functions, built only on eq-types, are exported to the opponent. The interface for decryption is however only exported to the ballot box.

$$\begin{aligned}\mathsf{dec\_ballot} :\ & c : \mathsf{eq\ bytes} \rightarrow \mathsf{privkey} \rightarrow m : \mathsf{bytes}\{|\forall z.\mathsf{Enc}(z,c) \Rightarrow z = m|\}\\\mathsf{check\_zkp} :\ & c : \mathsf{eq\ bytes} \rightarrow b : \mathsf{bool}\{|b = \mathsf{true} \Rightarrow (\mathsf{Valid}(c) \wedge (\exists m.\mathsf{Enc}(m,c))\wedge\\& \qquad (\mathsf{FromA}(c) \vee \mathsf{FromB}(c) \vee (\mathsf{FromO}(c) \wedge \mathsf{L}\ m = \mathsf{R}\ m)))|\}\\\mathsf{add\_ballot} :\ & c : \mathsf{eq\ bytes} \rightarrow c' : \mathsf{eq\ bytes} \rightarrow\\& c'' : \mathsf{eq\ bytes}\{|\forall m, m'.(\mathsf{Enc}(m,c) \wedge \mathsf{Enc}(m',c')) \Rightarrow \mathsf{Enc}(m+m',c'')|\}\end{aligned}$$

Intuitively, the type returned by decryption assures that the decryption of the ciphertext corresponds to the encrypted message. The successful application of the validity check on ballot c proves that the ballot is a valid encryption of either v_1 or v_2 and that it must come from either Alice, Bob, or the opponent. In the latter case it must be the same in both runs. When homomorphically adding two ciphertexts, the refinement of function add_ballot guarantees that the returned ciphertext contains the sum of the two. The implementation of dec_ballot is standard and consists of the application of the unsealing function. The implementation of check_zkp follows the approach proposed in [32, 33]: in particular, the zero-knowledge proof check function internally decrypts the ciphertexts and then checks the validity of the vote, returning a boolean value. Finally, the add_ballot homomorphic addition function is implemented in a similar manner, internally decrypting the two ciphertexts and returning a fresh encryption of the sum of the two plaintexts.

Global assumptions. In order to type-check the complete protocol we furthermore rely on three natural assumptions:

- A single ciphertext only corresponds to one plaintext, i.e., decryption is a function:

  $$\mathsf{assume}\ \forall m, m', c.(\mathsf{Enc}(m, c) \wedge \mathsf{Enc}(m', c)) \Rightarrow m = m'$$

- Alice and Bob only vote once:

  $$\mathsf{assume}\ \forall c, c'.(\mathsf{FromA}(c) \wedge \mathsf{FromA}(c')) \Rightarrow c = c'$$
  $$\mathsf{assume}\ \forall c, c'.(\mathsf{FromB}(c) \wedge \mathsf{FromB}(c')) \Rightarrow c = c'$$

Modeling revoting would require a bit more work. Revoting requires some policy that explains which ballot is counted, typically the last received one. In that case, we would introduce two types depending on whether the ballot is really the final one (there is a unique final one) or not.

3.4.4 Privacy analysis of Helios

Using the RF* type-checker (version 0.7.1-alpha) we have proved privacy for the homomorphic version of Helios. The corresponding files can be found in [88]. Our implementation builds on the above defined cryptographic library and global assumptions as well as Alice, Bob, and the ballot box BB as defined in the previous section.

We briefly give the intuition why the final tally $\mathit{result} = \mathsf{dec\_ballot}(b_{ABO})$ can be typed with type eq bytes, i.e., why both runs return the same value by explaining the typing of the ballot box BB.

- The ballots b_A, b_B, b_O that are received by the ballot box must have the following types (by definition of the corresponding ballot creation functions):

  $$\begin{aligned} &b_A : c : \mathsf{eq\ bytes}\{|\mathsf{Enc}(v_A, b_A) \wedge \mathsf{FromA}(b_A)|\} \\ &b_B : c : \mathsf{eq\ bytes}\{|\mathsf{Enc}(v_B, b_B) \wedge \mathsf{FromB}(b_B)|\} \\ &b_O : \mathsf{eq\ bytes} \end{aligned}$$

- Adding b_A and b_B together using add_ballot thus yields that the content of the combined ciphertext corresponds to $v_A + v_B$ and in particular, due to commutativity, this sum is the same in both protocol runs.

- The most significant effort is required to show that the payload v_O contained in b_O is indeed of type $\mathsf{eq\ bytes}$, i.e., $\mathsf{L}\ v_O = \mathsf{R}\ v_O$, meaning the sum of $v_A + v_B + v_O$ is the same in both runs. Intuitively, the proof works as follows: From checking the proof of b_O it follows that there exists v_O such that $\mathsf{Enc}(v_O, b_O) \wedge (\mathsf{FromA}(b_O) \vee \mathsf{FromB}(b_O) \vee (\mathsf{FromO}(b_O) \wedge \mathsf{L}\ v_O = \mathsf{R}\ v_O))$. From checking the distinctness of the ciphertexts we furthermore know that $b_A \neq b_O \neq b_C$. Given $\mathsf{FromA}(b_A)$ and $\mathsf{FromB}(b_B)$, the second and third global assumptions imply that neither $\mathsf{FromA}(b_O)$ nor $\mathsf{FromB}(b_O)$ hold true. Thus, it must be the case that $\mathsf{FromO}(b_O) \wedge \mathsf{L}\ v_O = \mathsf{R}\ v_O$.

3.5 Related work

Many symbolic protocol verification techniques have been applied to analyze e-voting systems [74, 76–79, 83, 86, 90]. In all of these works, the cryptographic primitives are modeled using terms and an equational theory, as opposed to the code-based abstractions we use in this chapter. While code-based abstractions of cryptography may look at a first glance more idealized than the modeling using equational theories, they are actually closer to ideal functionalities in simulation-based cryptographic frameworks. Although a formal computational soundness result is out of the scope of this work, the code-based abstractions we use are rather standard and computational soundness results for similar abstractions have been proven in [34, 55].

One of the main advantages of symbolic protocol verification is the potential for automation. However, current automated protocol verification tools are not yet mature enough to analyze most voting protocols. Firstly, existing tools do not support equational theories modeling homomorphic encryption. Thus, existing analyses of systems that rely on homomorphic tallying all rely on hand proofs [74, 86, 90], which are complicated and error-prone due to the complexity of the equational theories. Secondly, most current automated tools offer only limited support for verifying equivalence properties, which is required for verifying vote privacy. For instance, in [77] the analysis of Civitas using ProVerif relies on manual encodings and many other works, even though the equational theory is in the scope of the tools, again rely on hand proofs of observational equivalences [76, 83]. Although some recent tools, such as AKiSs [21] succeed in analyzing simple protocols such as [91], more complicated protocols are still out of reach. In [79], the privacy of the mix-net based version of Helios was shown using AKiSs, but mix-nets were idealized by simply outputting the decrypted votes in a non-deterministic order. In contrast, our model manipulates lists to express the fact that a mix-net produces a permutation. ProVerif was used to check some cases of verifiability [78], but automation is only supported for protocols without a homomorphic tally.

Other work on e-voting protocols considers definitions of privacy [92, 93] and verifiability [87, 94, 95] in computational models. However, no computer aided verification techniques have yet been applied in this context. Furthermore, prior work [96] demonstrated that individual and universal verifiability in general do not imply end-to-end verifiability, not even by assuming the no-clash property, using as an example the ThreeBallot voting system [97]. In this chapter, we have shown on the contrary that individual and universal verifiability do imply end-to-end verifiability. This is due to the fact that our individual verifiability notion is actually stronger and assumes that the board can be uniquely parsed as a list of ballots. This is the case for many voting systems but not for ThreeBallot where each ballot is split into three components.

3.6 Conclusion

In this chapter we proposed a novel approach, based on type-checking, for analyzing e-voting systems. It is based on a novel logical theory which allows to verify both verifiability and vote privacy, two fundamental properties of election systems. We were able to put this theory into practice and use an off-the-shelf type-checker to analyze the mix-net-, as well as homomorphic tallying-based versions of Helios, resulting in the first automated verification of Helios with homomorphic encryption. Indeed, the fact that the type-checker can discharge proof obligations on the algebraic properties of homomorphic encryption to an external solver is one of the strengths of this approach. Providing the right typing annotations constitutes the only manual effort required by our approach: in our analysis this was, however, quite modest, thanks to the support for type inference offered by RF*.

Part III

Type-Based Verification of Distributed Differential Privacy

4

DF7: A Type System for Distributed Differential Privacy

Differential privacy is a confidentiality property for database queries, which allows for the release of statistical information about the content of a database without disclosing personal data. The variety of database queries and enforcement mechanisms has recently sparked the development of a number of mechanized proof techniques for differential privacy.

Personal data, however, are often spread across multiple databases and queries have to be jointly computed by multiple, possibly malicious, parties. Many cryptographic protocols have been proposed to protect the data in transit on the network and to achieve differential privacy in a distributed, adversarial setting. Proving differential privacy for such protocols is hard and, unfortunately, out of the scope of the aforementioned mechanized proof techniques.

In this chapter, we present the first framework for the mechanized verification of distributed differential privacy. We propose a symbolic definition of differential privacy for distributed databases, which takes into account Dolev-Yao intruders and can be used to reason about compromised parties. Furthermore, we develop an affine, distance-aware type system to statically and automatically enforce distributed differential privacy in cryptographic protocol implementations (expressed in the RCF calculus). We also provide an algorithmic variant of our type system, which we prove sound and complete. Finally, we tested our analysis technique on a recently proposed protocol for privacy-preserving web analytics: we discovered a new attack acknowledged by the authors, proposed a fix, and successfully type-checked the revised variant.

Publication. In this chapter we extend the work © 2013 IEEE that was presented under the title 'Differential Privacy by Typing in Security Protocols' at the 26th IEEE Computer Security Foundations Symposium [7] in 2013.

4.1 Introduction

Personal information (e.g., patient records, browsing histories, social graphs, and behavioral data used for advertising) is disseminated in a wealth of databases spread across different institutions and services. On the one hand, disclosing information about these data is often desirable for improving services, analyzing trends, performing marketing studies, conducting research, and so on. On the other hand, this information leakage may irremediably compromise the privacy of users. Narayanan and Shmatikov [98] have shown that anonymized and aggregated data, which at first glance look seemingly harmless, may actually reveal an incredible amount of information on each user. The research community has long struggled to understand what privacy means in the context of database queries and how to measure the amount of information leaked by each query.

Differential privacy (DP). Today, differential privacy [99] is recognized as one of the fundamental notions of privacy for queries on statistical databases. Intuitively, a query is differentially private if it behaves statistically similarly on any pair of databases differing in one entry. In other words, the contribution of each single entry to the query result is bounded by a small constant factor, even if all remaining entries are known. A deterministic query can be made differentially private by perturbing the result with a certain amount of noise, thus reducing the accuracy of the answer.

Mechanized certification of differential privacy. Many works on differential privacy focus on specific classes of queries and noise mechanisms, proving properties thereof manually and in an ad-hoc manner. The disadvantage of this approach is that each new database type or each new query requires its own separate proof.

Taking up this challenge, recent research focused on the development of mechanized certification techniques for differential privacy. For instance, Reed and Pierce [36] showed how to automatically and statically enforce differential privacy for a large class of database queries based on a type system for a higher-order functional language. Gaboardi et al. [37] extended this type system to allow the certification of a larger class of queries whose sensitivity might depend on run-time information. Barthe et al. have presented CertiPriv [38], a mechanized framework based on interactive theorem proving that can be used to derive formal guarantees of differential privacy for a variety of sanitization mechanisms. In later work, Barthe et al. [39] have proposed an approach that verifies sanitization mechanisms with respect to the more general notion of approximative differential privacy using Hoare logic specifications. In a follow-up work [100], the authors present a relational refinement type system for the more general verification of mechanism design and approximative differential privacy. For a more detailed discussion of the related work, we refer to Section 4.10.

Distributed differential privacy (DDP). So far, we focused on the concept of differential privacy for single databases. In reality, data are often distributed across different databases, or stored on personal devices, and one has to compute statistical functions on the dataset obtained by joining these data, yet in a privacy-preserving manner. Since data may be read and manipulated by network attackers as well as compromised parties, these computations necessarily involve *cryptographic protocols*. A growing body of recent work has been focusing on the development of cryptographic protocols for distributed differential privacy (e.g., verifiable secret sharing [101], secure function evaluations [102], secure multi-party computations [103], multi-party distributed data aggregation algorithms [104–108], and local learning algorithms [109]) and on computational definitions of differential privacy against polynomially-bounded opponents [103]. So far, however, the differential privacy guarantees offered by such protocols had to be proven by hand.

Contributions. In this chapter, we introduce the first mechanized verification technique for distributed differential privacy, taking the first steps to reconcile the formal analysis of cryptographic protocols with the growing body of work on the formal certification of differential privacy guarantees. Specifically, we propose a symbolic definition of differential privacy for distributed databases. Our definition considers an attacker model that takes Dolev-Yao intruders[1] into account and can also be used to reason about compromised parties. Furthermore, we present an affine type system to statically enforce this privacy property in cryptographic protocol implementations. Our framework uniformly captures a variety of perturbation mechanisms, such as (discrete) Laplace noise addition [110, 111] and the exponential mechanism by McSherry and Talwar [112].

We also provide a sound and complete algorithmic variant of our type system called DF7$_{\text{alg}}$, which allows for automating the analysis. Finally, we tested our analysis technique on a protocol for privacy-preserving web analytics [11]: we discovered a new attack acknowledged by the authors, proposed a fix, and successfully type-checked the revised variant.

Outline. This chapter is organized as follows: Section 4.2 introduces the symbolic definition of distributed differential privacy. Section 4.3 presents the calculus. Section 4.4 explains the link between differential privacy and typing. Section 4.5 illustrates the DF7 type system. Section 4.6 explains the algorithmic variant DF7$_{\text{alg}}$ of the type system. Section 4.7 discusses our symbolic model of cryptography. Section 4.8 shows our type system at work on a cryptographic protocol for non-tracking web analytics. Section 4.9 shows how to extend the

[1] A Dolev-Yao intruder can overhear, intercept, and synthesize the cryptographic messages exchanged on the network, but it cannot break the cryptographic algorithms.

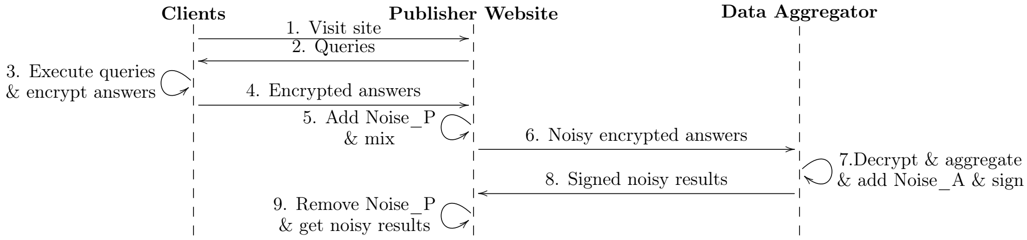


Figure 4.1: System for Non-Tracking Web Analytics [11]

type system to other sanitization mechanisms. Section 4.10 discusses the related work. Section 4.11 concludes the chapter and gives directions for future research.

For the proofs we refer to Appendix B. Section B.1 details the soundness proof of the DF7 system and Section B.2 presents the soundness proof of the algorithmic variant DF7$_{\text{alg}}$.

4.2 Distributed differential privacy

In this section we explain the key ideas behind our definition of distributed differential privacy. We also demonstrate the need for a mechanized proof technique for distributed differential privacy by showing a previously overlooked attack against a protocol on distributed databases.

4.2.1 Definition of differential privacy

A query is differentially private if it behaves statistically similarly on all databases D, D' differing in one entry, written $D \sim D'$. This means that the result of the query is not significantly changed by the presence or absence of each individual database entry.

The definition of differential privacy is parameterized by a number ϵ, which *measures* how strong the privacy guarantee is: the higher ϵ, the stronger the risk to join the database.

Definition 4.1 (Differential Privacy (ϵ-DP) [99])**.** *A randomized function f is* ϵ-differentially private *iff for all databases D, D' such that $D \sim D'$ and every set $S \subseteq Range(f)$,*

$$Pr[f(D) \in S] \leq e^{\epsilon} \cdot Pr[f(D') \in S]$$

A deterministic query can be made ϵ-differentially private by perturbing the result with a certain amount of noise, thus reducing the accuracy of the answer. We will describe some of these perturbation mechanisms throughout the chapter.

4.2.2 Definition of distributed differential privacy

As previously mentioned, data are frequently distributed across different databases or stored on user devices, and it is often desirable to compute statistical functions on the dataset obtained by joining these data. The exchange of data such as query results or fragments of a local database over an untrusted network necessarily involves cryptographic protocols.

Distributed differential privacy has to be defined with respect to the overall protocol and, particularly, against an opponent that can query the databases as well as interfere with the cryptographic messages exchanged on the network (e.g., by forging messages or replaying them).

The intuition underlying our definition of differential privacy is to think of the protocol as a query mechanism at disposal of the opponent. The database is distributed among protocol participants and, by interacting with them, the opponent can learn information about the database. The opponent is active in that it can choose how to interact with protocol participants based on the information acquired in the previous message exchanges. As usual in protocol analysis [31], we formalize the protocol as a function, which takes as input the database and can be called by the opponent. Typically, this function creates the cryptographic material, distributes the database across the protocol participants, and finally returns the functions implementing each of these parties. The opponent can call and schedule the execution of these functions at will.

Intuitively, a protocol P is *ϵ-differentially private* if for all databases D, D' differing in a single entry, the probability that the opponent outputs 1 when interacting with PD is approximately the same as when interacting with PD'.

Definition 4.2 (Distributed differential privacy (ϵ-DDP))**.** *P is* ϵ-differentially private *iff for all databases D, D' such that $D \sim D'$ and all opponents O,*

$$Pr[O(PD) \to^{*} 1] \leq e^{\epsilon} \cdot Pr[O(PD') \to^{*} 1]$$

This definition is the symbolic counterpart of the definition of ϵ_k-IND-CDP differential privacy for interactive protocols against polynomially-bounded opponents proposed by Mironov et al. [103]. Here the attacker is not polynomially bounded, since we work in a symbolic setting and under the perfect cryptography assumption: the Dolev-Yao attacker can only access the cryptographic libraries exported by the program, which model the idealized semantics of cryptographic primitives by constructs of the language and are, thus, suitable for automated verification.

Furthermore, we remark that the above definition can be used to reason about malicious parties, a common ingredient in the threat model of cryptographic protocols for distributed differential privacy, by simply letting these parties be under the control of the attacker (cf. Section 4.8).

Finally, notice that our definition can be used to reason about secrecy properties of cryptographic protocols in general, by simply letting the database be the set of secrets of protocol participants and by splitting the database so as to give each participant her own secret. In contrast to the existing symbolic definitions of secrecy for cryptographic protocols, which are based on reachability properties [113] or observational equivalence relations [114], this definition is *quantitative* in that it provides a bound on the amount of sensitive information leaked in a certain protocol execution.

4.2.3 What can go wrong?

Many new systems for distributed differential privacy are emerging. Analyzing the privacy properties of these systems by hand is not only tedious but, due to the complex nature of the systems and their underlying cryptographic protocols, also prone to overlooking attacks. For instance, we discovered a previously unknown attack on a recently proposed protocol for privacy-preserving web analytics by Akkus et al. [11].

The core part of this protocol is depicted in Figure 4.1. Intuitively, the provider of a website (*publisher*) uses a third party web analytics service (*data aggregator*) to gain aggregated information about the users visiting the site (*clients*). Such information can include user demographics (e.g., "how many men over 50 visited the website last week?"), browsing behavior, and information about the clients' systems. In order to achieve this goal in a privacy-preserving manner, the publisher acts as a proxy between the clients and the aggregator: it collects encrypted information sent by multiple clients visiting the website, mixes them with some fake "noisy" data, and forwards them to the aggregator. The aggregator decrypts all received ciphertexts and (not being able to distinguish between real and fake user information) combines the results. It adds noise to the produced analytics and forwards the "double noisy" results to the publisher.

Overall, both the publisher and the data aggregator obtain analytics about the clients that visited the publisher's webpage, but both results are not exact: the publisher cannot remove the noise that the aggregator added and vice-versa. Assuming that the noise mechanisms used by the publisher and aggregator to perturb the client data are differentially private and that the publisher and the data aggregator do not collude, one would assume that the overall protocol can be proved to enforce distributed differential privacy. As it turns out, this is not the case.

Consider the following new "snapshot" attack. Upon receiving the encrypted data of an individual client, a malicious publisher can duplicate these values multiple times and forward several copies either directly to the data aggregator, or to another publisher, thus allowing this client's data to influence the overall aggregated result multiple times.

This means that, even if a client stops answering queries about its personal data after its privacy budget[2] is spent, the publisher still possesses a "snapshot" of the previous answers and can replay them. The privacy budget is thus *not* under the control of the client, making it impossible to show ϵ-DDP for a fixed ϵ.

In the remainder of this chapter, we introduce a type system to statically enforce differential privacy in cryptographic protocol implementations. We use our type system to analyze the previously illustrated protocol in Section 4.8, where we give more details about the attack, propose a fix, and verify the resulting variant.

[2] *privacy budget*: the number of queries a client can safely answer in order to still guarantee ϵ-DDP

a	label
$h ::= \mathsf{inl} \mid \mathsf{inr} \mid \mathsf{fold}$	constructor
$M, N, D ::=$	value
x	variable
c	constant from Σ
f	function from Σ
(M, N)	multiplicative pair
$h\ M$	construction
$\lambda x.A$	function (x bound in A)
$\mathsf{read}_{a:\tau}$	reference read
$\mathsf{write}_{a:\tau}$	reference write
$A, B, P, Q ::=$	expression
M	value
$M\ N$	application
$\mathsf{let}\ x = A\ \mathsf{in}\ B$	let (x bound in B)
$\mathsf{let}\ (x, y) = M\ \mathsf{in}\ A$	split (x, y bound in A)
$\mathsf{case}\ M\ \mathsf{of}\ x\ \mathsf{in}\ A\ \mathsf{else}\ B$	case (x bound in A and B)
$\mathsf{unfold}\ M\ \mathsf{as}\ x\ \mathsf{in}\ A\ \mathsf{else}\ B$	unfold (x bound in A)
$M = N$	syntactic equality
ref_τ	reference creation
$\mathsf{add\_noise}^{s}_{\mathbb{Z}\to\mathbb{Z}}\ M$	discrete Laplace noise addition

Table 4.1: Syntax of RCF$_{\mathrm{DF7}}$

4.3 Review of RCF$_{\mathrm{DF7}}$

The λ-calculus used in this chapter (which we refer to as RCF$_{\mathrm{DF7}}$) is a dialect of RCF, a formal core of F# introduced by Bengtson et al. [31] to reason about cryptographic protocol implementations. Specifically, we adopt the computational variant proposed by Fournet et al. [34], which features mutable references and omits the fork operator. Contrary to RCF$_{\mathrm{AF7}}$, which we presented in Chapter 2, RCF$_{\mathrm{DF7}}$ lacks the assumption and assertion primitives present in previous RCF presentations [31,34], since they are only needed to specify policies but are of no semantic significance. We extend the calculus with a primitive for adding discrete Laplace noise [111], similar to the one for real-valued Laplace noise in the λ-calculus for differential privacy introduced by Reed and Pierce [36].

4.3.1 Syntax

The syntax of the calculus (cf. Table 4.1) includes standard functional *constructors*, *values*, and *expressions*. The language further supports discrete Laplace noise addition and references, by providing reference creators, readers, and writers. References are pairs of functions that read and write on a memory location. The execution of ref_τ allocates a fresh label a and returns a pair $(\mathsf{read}_{a:\tau}, \mathsf{write}_{a:\tau})$ of functions for reading and writing at location a. We statically annotate references with the type τ of the values stored therein. Note that type annotations do not have any semantic import.

The calculus and the type system are parameterized by a *signature* Σ, which exports constants, functions, and the respective types, as discussed in Section 4.4. This enhances the expressivity of our framework by making it extensible to new primitives. We assume that the constants exported by the signature include integers, ranged over by z, the unit value (), and sets, ranged over by S. We can encode boolean values as $\mathsf{true} \triangleq \mathsf{inl}\ ()$ and $\mathsf{false} \triangleq \mathsf{inr}\ ()$. The constructors some and none for option types can be encoded similarly. Lists can be encoded using the iso-recursive fold operator.

Remark (Discrete domains). Due to the physical limitations of actual machines we tacitly assume all types and the constants exported by the signature to range over discrete domains. For example, to include reals in the signature they must be expressed in some discrete fixed or floating point representation $\mathbb{R}_{disc}$ of $\mathbb{R}$. Whenever the context is clear we will just write $\mathbb{R}$.

4.3.2 Modeling cryptographic protocols

In order to model cryptographic protocols, we follow the approach proposed by Fournet et al. [34], for which computational soundness guarantees have been proved: the protocol is modeled as a function that is given to the opponent, who acts as a scheduler. In our setting such a protocol function expects a secret database as an input, splits the database entries amongst the n honest protocol participants, and returns n functions that model the behavior of each individual participant. If a participant is required to perform multiple message exchanges, the corresponding function is cascaded, expecting a message from the network as an input, and returning both the outgoing message and the code of the continuation function, which models the remaining protocol steps of that participant.

4.3.3 Semantics

We formalize the semantics of the calculus (cf. Table 4.2) using a labelled probabilistic reduction relation $[S, A] \xrightarrow{\ell}_p [S', A']$ between configurations $[S, A]$ consisting of a store S and an expression A. We track the probability p that a certain reduction takes place as well as the rule ℓ applied.

Non-deterministic reduction. The only non-deterministic primitive $\mathsf{add\_noise}^{s}_{\mathbb{Z}\to\mathbb{Z}}\ z_1$, which returns z_1+z_2, where z_2 is drawn according to the discrete Laplace distribution DLap^{s} (also known as the two-sided symmetric distribution) that has the probability mass function $Pr[x] = \frac{1-s}{1+s}s^{|x|}$ [111]. Following common practice in the literature on differential privacy [110], we use $Pr[x]$ to denote probability mass and probability density for both discrete and continuous random variables. In Section 4.4 we show that discrete Laplace noise addition is crucial to achieve ϵ-differential privacy. The label NOISE (z_1, z_2, s) keeps track of the arguments z_1, z_2 and of the parameter s of the discrete Laplace distribution.

$[S, (\lambda x.A)N] \xrightarrow{\mathsf{det}}_1 [S, A\{N/x\}]$	Red-Fun
$[S, \mathsf{let}\ (x,y) = (M,N)\ \mathsf{in}\ ]A \xrightarrow{\mathsf{det}}_1 [S, A\{M/x\}\{N/y\}]$	Red-Split
$[S, \mathsf{case\ inl}\ M\ \mathsf{of}\ x\ \mathsf{in}\ A\ \mathsf{else}\ B\ ] \xrightarrow{\mathsf{det}}_1 [S, A\{M/x\}]$	Red-CaseL
$[S, \mathsf{case\ inr}\ M\ \mathsf{of}\ x\ \mathsf{in}\ A\ \mathsf{else}\ B\ ] \xrightarrow{\mathsf{det}}_1 [S, B\{M/x\}]$	Red-CaseR
$[S, \mathsf{unfold\ fold}\ M\ \mathsf{as}\ x\ \mathsf{in}\ A\ \mathsf{else}\ B\ ] \xrightarrow{\mathsf{det}}_1 [S, A\{M/x\}]$	Red-Match
$[S, \mathsf{unfold}\ M\ \mathsf{as}\ x\ \mathsf{in}\ A\ \mathsf{else}\ B\ ] \xrightarrow{\mathsf{det}}_1 [S, B]$ if $\forall N.M \neq \mathsf{fold}\ N$	Red-Match-Fail
$[S, \mathsf{let}\ x = M\ \mathsf{in}\ A] \xrightarrow{\mathsf{det}}_1 [S, A\{M/x\}]$	Red-LetVal
$[S, \mathsf{let}\ x = A\ \mathsf{in}\ B] \xrightarrow{\ell}_p [S', \mathsf{let}\ x = A'\ \mathsf{in}\ B]$ if $[S, A] \xrightarrow{\ell}_p [S', A']$	Red-Let
$[S, \mathsf{ref}_\tau] \xrightarrow{\mathsf{det}}_1 [S \uplus \{a : \tau \mapsto \mathsf{none}\}, (\mathsf{read}_{a:\tau}, \mathsf{write}_{a:\tau})]$	Red-Ref
$[S \cup \{a : \tau \mapsto M\}, \mathsf{read}_{a:\tau}()] \xrightarrow{\mathsf{det}}_1 [S \cup \{a : \tau \mapsto \mathsf{none}\}, M]$	Red-Read
$[S \cup \{a : \tau \mapsto M\}, \mathsf{write}_{a:\tau} N] \xrightarrow{\mathsf{det}}_1 [S \cup \{a : \tau \mapsto \mathsf{some}\ N\}, ()]$	Red-Write
$[S, \mathsf{add\_noise}^{s}_{\mathbb{Z}\to\mathbb{Z}}\ z_1] \xrightarrow{\textsc{Noise}\ (z_1,z_2,s)}_{\mathsf{DLap}^s(z_2)} [S, z]$ where $z_1, z_2 \in \mathbb{Z}$ and $z = z_1 +_{\mathbb{Z}} z_2$	Red-Noise
$[S, M = M] \xrightarrow{\mathsf{det}}_1 [S, \mathsf{true}]$	Red-Eq-True
$[S, M = N] \xrightarrow{\mathsf{det}}_1 [S, \mathsf{false}]$ where $M \neq N$	Red-Eq-False
$[S, f\ F] \xrightarrow{\mathsf{det}}_1 [S, C]$ if $f(F) =_\Sigma C$	Red-Sig

Table 4.2: Semantics of RCF$_{\text{DF7}}$

Remark (Probabilistic semantics). Reed and Pierce [36] model perturbed results as distributions that are represented by monads and are given a deterministic, big-step denotational semantics. We found it more convenient to work with a probabilistic, small-step operational semantics, which is closer to the semantics traditionally used in type systems for cryptographic protocols and allows for leveraging existing proof techniques.

Deterministic reductions. The remaining deterministic primitives are labelled with det. The store is a finite map from references to values. The content of each reference can be either none, if the reference is empty, or some M. Departing from RCF, we define the semantics of references in a message-passing style, which is reminiscent of the concept of M-structures [115]: data are automatically removed from the referenced memory after being read, thus preventing data duplication. This choice simplifies the formalization of our affine type system and does not affect the expressivity of the language, since destructive and non-destructive read operators can be obtained from each other by encoding (e.g., non-destructive reads can be encoded by rewriting the read data).

Finally, the semantics of the calculus is parameterized by the semantics of the functions exported by Σ. These functions must be *deterministic*, take as argument *functional terms*, and return *constant terms*, which are defined below:

$$\begin{array}{lcll} F & := & f \mid c \mid (F_1, F_2) \mid h\ F & \text{functional term} \\ C & := & c \mid (C_1, C_2) \mid h\ C & \text{constant term} \end{array}$$

Notice that these functions may be higher-order and can be called by passing the arguments in uncurried form. Enforcing a clear separation between the functions exported by the signature and the values of our calculus is crucial to leverage existing results on function sensitivity and, thus, to make our framework easily extensible to new primitives and types. We write $f(F) =_\Sigma C$ to denote that C is the result of $f(F)$ in Σ and just use $f(F)$ to denote C whenever Σ is clear from the context.

Reduction probability. We finally define the probability that a certain expression P reduces into another expression Q, as required by the definition of ϵ-differential privacy stated in Section 4.1.

Definition 4.3 (Reduction probability)**.** *For all expressions P, Q and probabilities p, r, all $n \in \mathbb{N}^{>0}$, and all evaluation rules ℓ_i for $i \in [1, n]$,*

- *$P \xrightarrow{\ell_1,\ldots,\ell_n \;\; *}_{p\cdot r} Q$ iff there exist S, S', P' such that $[\emptyset, P] \xrightarrow{\ell_1,\ldots,\ell_{n-1} \;\; *}_{p} [S', P']$ and $[S', P'] \xrightarrow{\ell_n}_{r} [S, Q]$,*
- *$Pr[P \rightarrow^* Q] = \sum_{\ell_1,\ldots,\ell_n : P \xrightarrow{\ell_1,\ldots,\ell_n \;\; *}_{p} Q} p$.*

4.4 Differential privacy by typing

In this section we explain the intrinsic interplay between differential privacy and types in more detail.

As previously mentioned, a query can be made differentially private by perturbing the result with some noise. An important observation is that the amount of noise depends on the query: the more a single entry contributes to the query result, the stronger the noise has to be. The property we are interested in is the *sensitivity* of queries to quantitative differences in their inputs [110]. The sensitivity of a function measures how much this function amplifies the *distance* between inputs. Intuitively, queries of low sensitivity map nearby inputs to nearby outputs. The distance between values depends on their type: for the query below, the distance between databases is the Hamming distance, while the distance between real numbers is the Euclidean one. For instance, the query *"how many foreigners are registered in the hospital database?"* has sensitivity 1, since adding or removing a single entry will change the result by at most 1.

Reed and Pierce [36] showed there is an intimate connection between resource usage and function sensitivity: for instance, a deterministic function f that uses each entry of the database only k times and calls only 1-sensitive functions is at most k-sensitive, that is, it can magnify the distance of the inputs at most by a factor of k. Based on this intuition, they proposed an affine type system

$\tau, \rho ::= !_k\phi$		type ($k \in \mathbb{R}^{\geq 0} \cup \{\infty\}$)
$\phi, \psi ::=$		core type
	b	base type
	α	type variable
	$\mu\alpha.\tau$	iso-recursive type (α bound in τ)
	$\tau + \tau$	sum type
	$\tau \otimes \tau$	multiplicative pair type
	$\tau \multimap \tau$	function type

Convention: We write $!_k(!_{k'}\phi)$ to denote $!_{k \cdot k'}\phi$.

Table 4.3: Syntax of Types (DF7)

that statically bounds the usage of resources and, thus, can be used to statically over-approximate the sensitivity of a function and to enforce differential privacy.

4.4.1 Types

Table 4.3 shows the syntax of types. As usual in affine or linear type systems, the type of a value serves a double purpose: it describes the nature of the value (e.g., real number) as well as the number of times this value can be used at run-time. Consequently, the syntax of types is defined by mutual induction around the concept of *type* and *core type.*

Types of the form $!_k\phi$, with replication index k, describe values of core type ϕ that can be used *at most k* times at run-time. In this sense, our type system is affine, and thus more liberal than a linear type system would be, since it enforces an upper bound on the number of times a certain resource is used, as opposed to the exact number. In particular, if a function is given type $!_k\phi \multimap \tau$, then the argument can be used at most k times in the body of the function. Notice that values of type $!_\infty\phi$ can be used arbitrarily often and we call these types *exponential.* For the sake of readability, we often omit replication index $!_1$. Replication indexes are non-negative real numbers (i.e., $k \in \mathbb{R}^{\geq 0} \cup \{\infty\}$). Here a replication index of 0 denotes that a value cannot be used, while replication indexes k such that $0 < k < 1$ are helpful to express a sensitivity < 1. Core types comprise the base types b defined in the signature Σ, type variables, iso-recursive types, sum types, pair types, and function types. Note that the system by Reed and Pierce [36] includes additive pair types of the form $\tau \& \tau$: although we could easily add them to our system, we decided to omit them for simplicity, since they are not useful for the examples we considered.

4.4.2 Distance on types

We require a notion of distance for all types and core types in our type system. Note that this means that the signature has to provide a metric for all base types. We adopt the metric for types introduced by Reed and Pierce [36], which we overview below. For types with replication index k, we define the distance as the distance of the core type multiplied by k, thus $\delta_{!_k\phi}(x, y) = k \cdot \delta_\phi(x, y)$. For core pair types, the distance is defined as the sum of the distances of the components $\delta_{\tau\otimes\rho}((x_1,x_2),(y_1,y_2)) = \delta_\tau(x_1,x_2)+\delta_\rho(y_1,y_2)$. The distance of functions is defined as the maximal distance of the outputs that the two functions produce for the same input $\delta_{\tau\multimap\rho}(f,g) = \mathsf{max}_{x\in\tau}(\delta_\rho(f(x),g(x)))$. The distance on core sum types is defined as

$$\delta_{\tau+\rho}(x,y) = \begin{cases} \delta_\tau(x',y') & \text{if } x = \mathsf{inl}\ x' \text{ and } y = \mathsf{inl}\ y' \\ \delta_\rho(x',y') & \text{if } x = \mathsf{inr}\ x' \text{ and } y = \mathsf{inr}\ y' \\ \infty & \text{otherwise} \end{cases}$$

Intuitively, $\mathsf{inl}\ x$ and $\mathsf{inr}\ x$ have distance ∞ since one can perform a pattern-matching operation on them and, based on the result, produce arbitrarily distant results. The distance of two values of an iso-recursive core type is intuitively defined by unfolding the two values. Note that the following definition $\delta_{\mu\alpha.\tau}(\mathsf{fold}\ x, \mathsf{fold}\ y) = \delta_{\tau\{\mu\alpha.\tau/\alpha\}}(x,y)$ of distance for iso-recursive types is not well-founded in all cases (e.g., for the type $\mu\alpha.\alpha$). We do not consider types for which this distance is not well-founded, as they are not needed in any practical example we considered.

Example 4.1. $\delta_{!_3(\mathbb{R}\otimes\mathbb{R})}((1,2),(0,3)) = 3\cdot(1+1) = 6$ *and* $\delta_{\mathbb{R}\multimap\mathbb{R}}(\lambda x.x, \lambda x.x+1) = 1$ *and* $\delta_{\mathbb{R}\multimap\mathbb{R}}(\lambda x.x, \lambda x.x + x) = \infty$.

4.4.3 Signature

As mentioned in Section 4.3, our framework is parameterized by a signature Σ, which exports constants, functions, and the respective types. In this chapter, we assume that integers are given type $\mathbb{Z}$, the unit value type Unit, while sets of values of type τ are given type $\mathsf{Set}\langle\tau\rangle$. The signature additionally needs to provide a distance δ_b for each base type b: $\delta_\mathbb{Z}$ is the Euclidean distance between integers, δ_{Unit} is the null distance, while $\delta_{\mathsf{Set}\langle\tau\rangle}$ is the symmetric difference (the number of entries that are contained in one but not in the other set). Given these types, we can define standard encodings for the (core) types Bool, $\mathsf{Option}\langle\tau\rangle$, and $\mathsf{List}\langle\tau\rangle$, as shown below:

$\mathsf{Bool} \triangleq !_\infty\mathsf{Unit}+!_\infty\mathsf{Unit}$	boolean type
$\mathsf{Option}\langle\tau\rangle \triangleq !_\infty\mathsf{Unit} + \tau$	option type
$\mathsf{List}\langle\tau\rangle \triangleq \mu\alpha.!_1(!_\infty\mathsf{Unit} + (!_1(\tau\otimes!_1\alpha)))$	list type

4.4.4 Type-based k-sensitivity and differential privacy

The notion of sensitivity can be generalized to arbitrary types τ for which a metric δ_τ exists, as shown below:

Definition 4.4 (Type-Based k-Sensitivity [36])**.** *A function f is k-sensitive in $\tau_1 \to \tau_2$ iff $\delta_{\tau_2}(f(x),f(y)) \leq k\cdot\delta_{\tau_1}(x,y)$ for all $x,y \in \tau_1$.*

Example 4.2. *The function $f_1(x,y) = x+y$ is 1-sensitive in $(\mathbb{Z}\otimes\mathbb{Z}) \to \mathbb{Z}$. Notice that each argument is used only once. The function $f_2(x) = 3x$ has sensitivity 3* *in $\mathbb{Z} \to \mathbb{Z}$. One might say that here the argument is used only once, but since multiplication is defined using addition, $f_2(x) = x+x+x$ in fact uses the argument three times.*

The sensitivity of a function determines the amount of noise that one has to add to the result for rendering the function in question ϵ-differentially private.

Proposition 4.1 (Sensitivity and DP [116])**.** *Suppose f is k-sensitive in $\mathcal{D}^n \to \mathbb{Z}$. Define the random function $q = \lambda x.\mathsf{add\_noise}^{e^{-\epsilon/k}}_{\mathbb{Z}\to\mathbb{Z}}\ f(x)$. Then q is ϵ-differentially private.*

Notice that k-sensitivity can be reduced to 1-sensitivity.

Proposition 4.2 (k-Sensitivity vs 1-Sensitivity [36])**.** *A function f is k-sensitive in $\tau_1 \to \tau_2$ if and only if it is a 1-sensitive function in $!_k\tau_1 \to \tau_2$.*

Example 4.3. *The function f_2 defined in Example 4.2 is 1-sensitive in type $!_3\mathbb{Z} \to \mathbb{Z}$.*

In the type system by Reed and Pierce, $!_k\tau_1 \multimap \tau_2$ is the type of k-sensitive functions in $\tau_1 \to \tau_2$ (or, equivalently, 1-sensitive functions in $!_k\tau_1 \to \tau_2$). This holds true for deterministic functions in our type system and, in the next section, we generalize this property to stateful randomized expressions.

4.5 The DF7 type system

In this section, we introduce a distance-aware type system that enforces ϵ-differential privacy in cryptographic protocol implementations. We will refer to the system as DF7. The key idea is to enforce a *distance preservation* property for well-typed configurations, which is close in spirit to the notion of sensitivity for deterministic functions but additionally takes into account the store and the reduction probabilities. Given two configurations that share the same structure and only differ in some of their constants, their distance is defined by summing the distance of the differing constants. Intuitively, the distance preservation property says that two well-typed configurations sharing the same structure reduce with approximately the same probability into configurations whose distance does not exceed the distance of the initial configurations.

4.5.1 Typing environment and judgments

The definition of typing environments and the judgements of DF7 are listed in Table 4.4. The typing environment is a list of type bindings for variables, kind bindings for type variables, and type variable declarations of the form $x : \tau, \alpha :: \kappa$, and α, respectively. The type system comprises six typing judgments: the well-formedness judgments $\Gamma \vdash \diamond$ and $\Gamma \vdash \tau$ for typing environments and types, respectively; the kinding judgments $\Gamma \vdash \tau :: \kappa$ and $\Gamma \vdash \phi :: \kappa$, which classify types and core types based on whether the corresponding values may be sent to the

$\mu ::=$		environment entry
	$x : \tau$	type binding
	$\alpha :: \kappa$	kind binding ($\kappa \in \{\mathsf{pub}, \mathsf{tnt}\}$)
	α	type variable

$\Gamma, \Delta ::= \emptyset \mid \Gamma, \mu$	environment
$\Gamma \vdash \diamond$	well-formedness of environments
$\Gamma \vdash \tau$	well-formedness of types
$\Gamma \vdash \phi :: \kappa$	kinding for core types
$\Gamma \vdash \tau :: \kappa$	kinding for types
$\Gamma \vdash \tau <: \rho$	subtyping
$\Gamma \vdash A : \tau$	typing of expressions

Table 4.4: Typing environment and judgements (DF7)

(Type)
$$\frac{\Gamma \vdash \diamond \qquad \mathit{ft}(\theta) \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash \theta}$$

(Env Empty)
$$\frac{}{\emptyset \vdash \diamond}$$

(Env Entry)
$$\frac{\Gamma \vdash \diamond \qquad \mathit{ft}(\mu) \subseteq \mathit{dom}(\Gamma) \qquad \mathit{dom}(\mu) \cap \mathit{dom}(\Gamma) = \emptyset}{\Gamma, \mu \vdash \diamond}$$

Notation: We write θ to denote both types and core types.

Table 4.5: Well-formedness judgments (DF7)

opponent (kind pub) or be received from the opponent (kind tnt); the subtyping judgment $\Gamma \vdash \tau <: \rho$, which enhances the expressivity of the type system by allowing values of type τ to be used in place of values of type ρ; and the typing judgment for expressions $\Gamma \vdash A : \tau$.

4.5.2 Well-formedness judgments

The well-formedness rules are stated in Table 4.5. A type τ is well-formed in Γ if the type variables occurring free in τ are bound in Γ. A typing environment Γ is well-formed if the variables bound in Γ are all distinct and their types are well-formed.

The domain of environments is defined as $\mathit{dom}(\emptyset) \triangleq \emptyset$, $\mathit{dom}(\Gamma, \alpha) \triangleq \mathit{dom}(\Gamma) \cup \{\alpha\}$, $\mathit{dom}(\Gamma, \alpha :: \kappa) \triangleq \mathit{dom}(\Gamma) \cup \{\alpha\}$, and $\mathit{dom}(\Gamma, x : \tau) \triangleq \mathit{dom}(\Gamma) \cup \{x\}$. The set of free type variables in $\tau =!_k\phi$ is denoted by $\mathit{ft}(\tau) \triangleq \mathit{ft}(\phi)$, where $\mathit{ft}(x : \tau) \triangleq \mathit{ft}(\tau)$ and $\mathit{ft}(\alpha) \triangleq \emptyset$.

4.5.3 Kinding and subtyping

As we show in Chapter 2, one of the standard ingredients of type systems for cryptographic protocols is the kinding relation [31–33]: a type has kind *public* if

$$\text{(Kind Sig)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash b :: \kappa} \qquad \text{(Kind Sum)}\ \frac{\begin{array}{c}\Gamma \vdash \tau :: \kappa \\ \Gamma \vdash \rho :: \kappa\end{array}}{\Gamma \vdash \tau + \rho :: \kappa} \qquad \text{(Kind Pair)}\ \frac{\Gamma \vdash \tau :: \kappa \qquad \Gamma \vdash \rho :: \kappa}{\Gamma \vdash \tau \otimes \rho :: \kappa} \qquad \text{(Kind Fun)}\ \frac{\Gamma \vdash \tau :: \overline{\kappa} \qquad \Gamma \vdash \rho :: \kappa}{\Gamma \vdash \tau \multimap \rho :: \kappa}$$

$$\text{(Kind Var)}\ \frac{\alpha :: \kappa \in \Gamma \qquad \Gamma \vdash \diamond}{\Gamma \vdash \alpha :: \kappa} \qquad \text{(Kind Rec)}\ \frac{\Gamma, \alpha :: \kappa \vdash \tau :: \kappa}{\Gamma \vdash \mu\alpha.\tau :: \kappa} \qquad \text{(Kind Tnt)}\ \frac{\Gamma \vdash \phi :: \mathsf{tnt}}{\Gamma \vdash !_k\phi :: \mathsf{tnt}} \qquad \text{(Kind Pub)}\ \frac{\Gamma \vdash \phi :: \mathsf{pub}}{\Gamma \vdash !_\infty\phi :: \mathsf{pub}}$$

$$\text{(Sub Kind)}\ \frac{\Gamma \vdash \tau :: \mathsf{pub} \qquad \Gamma \vdash \rho :: \mathsf{tnt}}{\Gamma \vdash \tau <: \rho} \qquad \text{(Sub Repl)}\ \frac{\Gamma \vdash !_1\phi <: !_1\psi \qquad k \leq t}{\Gamma \vdash !_t\phi <: !_k\psi} \qquad \text{(Sub Refl)}\ \frac{\Gamma \vdash \tau}{\Gamma \vdash \tau <: \tau}$$

$$\text{(Sub Sum)}\ \frac{\Gamma \vdash \tau <: \tau' \qquad \Gamma \vdash \rho <: \rho'}{\Gamma \vdash !_1(\tau + \rho) <: !_1(\tau' + \rho')} \qquad \text{(Sub Pair)}\ \frac{\Gamma \vdash \tau <: \tau' \qquad \Gamma \vdash \rho <: \rho'}{\Gamma \vdash !_1(\tau \otimes \rho) <: !_1(\tau' \otimes \rho')}$$

$$\text{(Sub Fun)}\ \frac{\Gamma \vdash \tau' <: \tau \qquad \Gamma \vdash \rho <: \rho'}{\Gamma \vdash !_1(\tau \multimap \rho) <: !_1(\tau' \multimap \rho')} \qquad \text{(Sub Pos Rec)}\ \frac{\Gamma, \alpha \vdash \tau <: \rho \qquad \alpha \text{ occurs only positively in } \tau \text{ and } \rho}{\Gamma \vdash !_1(\mu\alpha.\tau) <: !_1(\mu\alpha.\rho)}$$

Notation: $\overline{\mathsf{pub}} = \mathsf{tnt}$ and $\overline{\mathsf{tnt}} = \mathsf{pub}$.

Table 4.6: Kinding and subtyping relations (DF7)

messages of that type can be sent to the opponent and kind *tainted* if messages of that type can be received from the opponent. The types that are both public and tainted are equivalent by subtyping (i.e., they are subtypes of each other). This is crucial to prove the opponent typability lemma which says that all opponents are well-typed. This property allows us to assume that the attacker is well-typed in the proof of soundness without fearing that our typing discipline limits the power of the attacker. A property of our type system is that a type is both public and tainted if and only if all replication indexes occurring therein are set to ∞. We call such types *opponent types*. Formally, we write $[\tau]_\infty$ to denote the type obtained by replacing all replication indexes in τ with ∞ and we define the set $\mathcal{OPP}$ of opponent types as the set of types τ such that $\tau = [\tau]_\infty$.

The kinding and subtyping relations of DF7 are reported in Table 4.6. The rules are mostly standard [31]. The kind of type variables is determined by the typing environment Γ (Kind Var). A type is public only if it has the form $!_\infty\phi$, since the opponent can use the received values arbitrarily often, and ϕ is public (Kind Pub). A type of the form $!_k\phi$ is tainted only if ϕ is tainted (Kind Tnt): we do not place any constraint on k, since we can choose to limit the number of times our code is allowed to use the values received from the opponent. The core types defined in Σ are both public and tainted, since the constants in the signature can be used by the opponent (Kind Sig). A sum or pair type is public only if both of its components are public and tainted if both components are tainted (Kind Sum) and (Kind Pair) respectively. A function type is public if

$$\begin{array}{rcl}
\emptyset + \emptyset & = & \emptyset \\
(\Gamma, \alpha) + (\Delta, \alpha) & = & (\Gamma + \Delta), \alpha \\
(\Gamma, \alpha :: \kappa) + (\Delta, \alpha :: \kappa) & = & (\Gamma + \Delta), \alpha :: \kappa \\
(\Gamma, x :!_k\tau) + (\Delta, x :!_{k'}\tau) & = & (\Gamma + \Delta), x :!_{k+k'}\tau
\end{array}$$

Notation: $k\Gamma$ denotes the environment obtained by multiplying all replication indexes of the type bindings in Γ by a factor k

Table 4.7: Environment sum (DF7)

it outputs public values and accepts tainted values as inputs and tainted in the opposite case (Kind Fun). The kinding of an iso-recursive type is determined by recursively kinding under an environment extended with the binding of the type variable to the expected kind (Kind Rec).

Similarly, subtyping is mostly standard: the subtyping relation is reflexive (Sub Refl), sum and pair types are covariant in their components (Sub Sum) and (Sub Pair), iso-recursive types are subtyped recursively if the type variable occurs only positively in both types (Sub Pos Rec), and function types are contravariant in their inputs and covariant in their outputs (Sub Fun). The only aspect that is worth pointing out is that the subtyping relation is contravariant in the replication index (Sub Repl), i.e., $\Gamma \vdash !_k\tau <:!_j\tau$ only if $j \leq k$. In particular, this means that values of type $!_k\tau$ for $k < \infty$ can never be sent to the opponent, since public types have replication index ∞ and the replication index can never be increased by subtyping. This is crucial for the soundness of the type system, since the opponent can use any value it receives arbitrarily often, in particular $k + 1$ times. However, values received from the attacker at type $!_\infty\tau$ can be super-typed to $!_j\tau$ and be treated as confidential.

4.5.4 Typing values

One of the goals of our type system is to statically enforce that values of type $!_k\tau$ are used at most k times at run-time. Intuitively, this is achieved by formalizing the typing derivations for values and expressions in a way that the environment used in the thesis is the sum of the environments used in the hypotheses (or reading the typing rules upside down, the environment is split along the typing derivations). This prevents multiple usages of the same resource.

The sum of environments is formalized in Table 4.7. Type variables and kind bindings are not subject to cardinality constraints, since they are not associated to any values.[3] If the two typing environments contain type bindings of the form $x :!_k\tau$ and $x :!_{k'}\tau$, respectively, their sum returns an environment containing $x :!_{k+k'}\tau$.

The rules for typing values are reported in Table 4.8 and are mostly standard [36]. Variables are given the type they are bound to in the typing environ-

[3] Indeed, the reason why we do not need to add the environments along the kinding and subtyping derivations is that they only depend on the type variables and kind bindings in the environment.

$$\textsc{(Var)}\ \frac{\Gamma \vdash \diamond \quad x :!_k\phi \in \Gamma \quad k \geq 1}{\Gamma \vdash x :!_1\phi} \qquad \textsc{(Sig)}\ \frac{M : \phi \in \Sigma \quad M \in \{c, f\} \quad \Gamma \vdash \diamond \quad \emptyset \vdash \phi}{\Gamma \vdash M :!_1\phi} \qquad (\otimes\mathrm{I})\ \frac{\Delta \vdash M_1 : \tau_1 \quad \Gamma \vdash M_2 : \tau_2}{\Delta + \Gamma \vdash (M_1, M_2) :!_1(\tau_1 \otimes \tau_2)}$$

$$(+/\mu\mathrm{I})\ \frac{h : (\tau, \phi) \quad \Gamma \vdash M : \tau \quad \Gamma \vdash \phi}{\Gamma \vdash h\ M :!_1\phi} \qquad (\multimap\mathrm{I})\ \frac{\Gamma, x : \tau \vdash A : \rho}{\Gamma \vdash \lambda x.A :!_1(\tau \multimap \rho)} \qquad \textsc{(Read)}\ \frac{\Gamma \vdash \tau}{\Gamma \vdash \mathsf{read}_{a:\tau} :!_1\mathsf{Read}\langle\tau\rangle}$$

$$\textsc{(Read Opp)}\ \frac{\Gamma \vdash \tau \quad \tau \in \mathcal{OPP}}{\Gamma \vdash \mathsf{read}_{a:\tau} :!_1[\mathsf{Read}\langle\tau\rangle]_\infty} \qquad \textsc{(Write)}\ \frac{\Gamma \vdash \tau}{\Gamma \vdash \mathsf{write}_{a:\tau} :!_1\mathsf{Write}\langle\tau\rangle} \qquad (!_k\mathrm{I})\ \frac{\Gamma \vdash M :!_1\phi \quad k > 0}{k\Gamma \vdash M :!_k\phi}$$

Notation:

$$\begin{array}{lcl}
\mathsf{inl} & : & (\tau, \tau + \tau') \\
\mathsf{inr} & : & (\tau, \tau' + \tau) \\
\mathsf{fold} & : & (\tau\{\mu\alpha.\tau/\alpha\}, \mu\alpha.\tau) \\
\mathsf{Read}\langle\tau\rangle & \triangleq & !_\infty\mathsf{Unit} \multimap !_1\mathsf{Option}\langle\tau\rangle \\
\mathsf{Write}\langle\tau\rangle & \triangleq & \tau \multimap !_\infty\mathsf{Unit}
\end{array}$$

Table 4.8: Typing rules for values (DF7)

ment Γ under the condition that the corresponding replication index is at least 1. For typing pairs, one has to type each of the components individually and sum the typing environments, thus avoiding duplication of resources ($\otimes$I). Sum and iso-recursive constructors are typed recursively using rule ($+/\mu$I). Functions are typed by typing the body of the function under an environment extended with the binding of the argument variable to the expected argument type ($\multimap$I). As discussed in Section 4.4, the signature Σ defines a core type ϕ for each constant and function. These are given the core type specified in Σ with an affine replication factor (Sig). The function $\mathsf{read}_{a:\tau}$ to read from reference a is given type $!_1(!_\infty\mathsf{Unit} \multimap !_1\mathsf{Option}\langle\tau\rangle)$ (Read), since it takes as input the unit value and returns either $\mathsf{some}\ M$, if M is the value currently stored in the reference, or none, if the reference is empty. Rule (Read Opp) applies to references whose content is not subject to any cardinality constraint (i.e., $\tau \in \mathcal{OPP}$): in this case, the value read from the reference can be used arbitrarily often and the replication index of the option type can thus be ∞. This rule is needed to type-check the opponent. The function $\mathsf{write}_{a:\tau}$ is instead given type $!_1(\tau \multimap !_\infty\mathsf{Unit})$, since it takes as input a value of type τ to be stored in the reference and returns the unit value (Write). Finally, we can introduce types with arbitrary positive replication index using rule ($!_k$I): if M is given type $!_1\phi$ under Γ, then it is possible to type-check M with type $!_k\phi$ given the environment $k\Gamma$.

$$\text{(Sub)}\quad \frac{\Gamma \vdash A : \tau \qquad \Gamma \vdash \tau <: \rho}{\Gamma \vdash A : \rho}$$

$$(\otimes\text{E})\quad \frac{\Delta \vdash M :!_1(\tau_1 \otimes \tau_2) \qquad \Gamma, x :!_r\tau_1, y :!_r\tau_2 \vdash A : \tau'}{r\Delta + \Gamma \vdash \mathsf{let}\ (x, y) = M\ \mathsf{in}\ A : \tau'}$$

$$(+\text{E})\quad \frac{\Gamma \vdash M :!_1(\tau_1 + \tau_2) \qquad \Delta, x :!_r\tau_1 \vdash A : \tau' \qquad \Delta, x :!_r\tau_2 \vdash B : \tau'}{\Delta + r\Gamma \vdash \mathsf{case}\ M\ \mathsf{of}\ x\ \mathsf{in}\ A\ \mathsf{else}\ B\ \ : \tau'}$$

$$(\mu\text{E})\quad \frac{\Gamma \vdash M :!_1\mu\alpha.\tau \qquad \Delta, x :!_r\tau\{\mu\alpha.\tau/\alpha\} \vdash A : \tau' \qquad \Delta \vdash B : \tau'}{\Delta + r\Gamma \vdash \mathsf{unfold}\ M\ \mathsf{as}\ x\ \mathsf{in}\ A\ \mathsf{else}\ B\ \ : \tau'}$$

$$(\multimap\text{E})\quad \frac{\begin{array}{c}\Delta \vdash M_1 :!_1(\tau \multimap \tau') \\ \Gamma \vdash M_2 : \tau\end{array}}{\Delta + \Gamma \vdash M_1 M_2 : \tau'}$$

$$\text{(Let)}\quad \frac{\Gamma \vdash A : \tau \qquad \Delta, x : \tau \vdash B : \tau'}{\Delta + \Gamma \vdash \mathsf{let}\ x = A\ \mathsf{in}\ B : \tau'}$$

$$\text{(Add-Noise)}\quad \frac{\Gamma \vdash M :!_1\mathbb{Z}}{\Gamma \vdash \mathsf{add\_noise}^{s}_{\mathbb{Z}\to\mathbb{Z}}\ M :!_\infty\mathbb{Z}}$$

$$\text{(Ref)}\quad \frac{\Gamma \vdash \tau}{\Gamma \vdash \mathsf{ref}_\tau :!_\infty\mathsf{Ref}\langle\tau\rangle}$$

$$\text{(Ref Opp)}\quad \frac{\Gamma \vdash \tau \qquad \tau \in \mathcal{OPP}}{\Gamma \vdash \mathsf{ref}_\tau : [!_\infty\mathsf{Ref}\langle\tau\rangle]_\infty}$$

$$\text{(Eq)}\quad \frac{\Gamma \vdash M :!_\infty\tau \qquad \Delta \vdash N :!_\infty\tau}{\Gamma + \Delta \vdash M = N :!_\infty\mathsf{Bool}}$$

Notation: $\mathsf{Ref}\langle\tau\rangle \triangleq !_\infty\mathsf{Read}\langle\tau\rangle \otimes !_\infty\mathsf{Write}\langle\tau\rangle$

Table 4.9: Typing rules for expressions (DF7)

4.5.5 Typing expressions

The typing rules for expressions are reported in Table 4.9. The subsumption rule (Sub) is standard since subtyping does not rely on the addition of environments. The rule for eliminating function types ($\multimap$E) is straightforward, checking that the argument is given the expected function argument type and returning the result type. In the case of standard let expressions (Let), the variable x is bound to the type of expression A and expression B is typed under the environment that is extended with this binding. The rules for eliminating pair types ($\otimes$E), sum types (+E), and iso-recursive types (μE) are standard [36], adding the environments created along the subderivations. Note that in these cases, the variable x (and y respectively) is assumed to be used with replication factor r in the scope of the elimination. The cost of this choice is that the environment Γ that was used to type the value M must be multiplied by the same factor r. The $\mathsf{add\_noise}^{s}_{\mathbb{Z}\to\mathbb{Z}}\ M$ expression is given type $!_\infty\mathbb{Z}$ (Add-Noise), since the amount of noise added to M suffices to hide any dependency on secret data. We will see later on that the noise parameter s has to correspond to the replication index of the database type in order to achieve ϵ-DDP. The equality check between two values M, N is well-typed with type $!_\infty\mathsf{Bool}$ if the two values are of the same exponential type $!_\infty\tau$ (Eq). The constraint on the replication factor is crucial to enforce distance preservation, as the possible results false and true of an equality check have distance ∞. Finally, the expression ref_τ creates a fresh reference and returns the pair of functions to read from and write into this reference: the type of this expression is thus obtained by pairing the types of the reader and writer (Ref). Similar to the case of the typing of the reader function (Read Opp), in the case that $\tau \in \mathcal{OPP}$, the reference can also be given replication factor ∞ throughout by applying the opponent typing rule (Ref Opp).

4.5.6 Formal results

In the following section we show the main soundness results of DF7. For the proofs we refer to Section B.1.

The proof of the differential privacy theorem relies on an opponent typability lemma, saying that all opponents are well-typed. As usual in type systems for cryptographic protocols [31], we require the opponent to be a closed expression that is annotated only with an untrusted type (Un in the literature, $\tau \in \mathcal{OPP}$ in our case), which characterizes the values that can be sent to and received from the opponent. Note that we are not constraining the opponent, since typing annotations do not affect the semantics of expressions.

Our main theorem uses the following generalized definition of distributed differential privacy that considers constant terms D, D' of arbitrary type and arbitrary distance as inspired by Reed and Pierce [36].

Definition 4.5 (Generalized ϵ-DDP (ϵ, τ-DDP))**.** *P is ϵ, τ-*differentially private *iff for all constant terms D, D' of type τ and all opponents O,*

$$Pr[O(PD) \to^* 1] \leq e^{\epsilon \cdot \delta_\tau(D,D')} \cdot Pr[O(PD') \to^* 1]$$

Theorem 4.1 below states that all well-typed expressions are ϵ, τ-differentially private. Note that we actually prove a more general property, parameterized by the noise added in the protocol execution.

Theorem 4.1 ($(\epsilon/k, \tau)$-Differential Privacy)**.** *For all $k \in \mathbb{R}^{>0}$, all types τ, and all closed expressions P such that the following conditions hold:*

- *the parameter of all noise addition primitives is set to $e^{-\epsilon/k}$ (i.e., they are of the form $\mathsf{add\_noise}^{e^{-\epsilon/k}}_{\mathbb{Z}\to\mathbb{Z}}\ M$)*
- *$\emptyset \vdash P : \tau \multimap \rho$ for some $\rho \in \mathcal{OPP}$*

P is $\epsilon/k, \tau$-differentially private.

We also remark that DF7 can be used to enforce a strong secrecy property [34] based on probabilistic observational equivalence for expressions that do not contain any noise addition primitives, as formalized below.

Corollary 4.1 (Strong Secrecy)**.** *For all opponents O, all types τ, and all closed expressions P such that the following conditions hold:*

- *P does not contain any noise addition primitives*
- *$\emptyset \vdash P : \tau \multimap \rho$ for some $\rho \in \mathcal{OPP}$*

we have that $Pr[O(PD) \rightarrow^* 1] = Pr[O(PD') \rightarrow^* 1]$

Finally, one might wonder how this approach scales to multiple protocol sessions. For example, let us consider a protocol that is given type $\tau \multimap \rho$, where $\rho \in \mathcal{OPP}$ in which the parameter of all noise addition primitives is $e^{-\epsilon}$. By Theorem 4.1, this protocol achieves ϵ, τ-DDP. We might want to allow this protocol to be executed i times. Intuitively, such a multisession protocol will have type $!_i\tau \multimap \rho$, since the secret database will have to be accessed i times, and will thus be $\epsilon, !_i\tau$-differentially private. This means that the ratio between the probability of outputting 1 when the protocol is initialized with database D and the probability of outputting 1 when the protocol is initialized with D' is bounded by $e^{\epsilon \cdot \delta_{!_i\tau}(D,D')}$. By the distance definition, this is equivalent to $e^{\epsilon \cdot i \cdot \delta_\tau(D,D')}$. This means that the multisession protocol is $(\epsilon \cdot i), \tau$-differentially private. Intuitively, privacy losses are summed up over multiple sessions (or queries), as stated by the principle of composition formulated by McSherry [117].

4.6 Algorithmic type-checking (DF7$_{\text{alg}}$)

The treatment of type bindings, as formulated in Section 4.5, results in several non-deterministic rules. This non-determinism complicates the implementation of an efficient decision procedure. In the following section, we present a sound and complete algorithmic version of the type system.

4.6.1 Key ideas

It is well-known that standard sources of non-determinism like subtyping and typing constructors of sum or iso-recursive types can be resolved using type annotations.

We first focus on the distinctive source of non-determinism of DF7, the splitting of typing environments, which is much harder to deal with in an algorithmic way, before providing the full details of the algorithmic type system DF7$_{\text{alg}}$.

The core idea underlying the algorithmic version of the type system is inspired by the work of Cervesato et al. [118] on efficient resource management for linear logic proof search. Intuitively, the algorithmic variant of a typing rule that relies on the splitting of the typing environment among its premises proceeds as follows: all resources (i.e., type bindings) are first used to prove the first premise of a typing rule; the resources that were not consumed in the derivation of that premise are then returned and used in the derivation of the second premise and so on.

Algorithmic typing judgements are of the form $\Gamma \vdash_{\mathsf{alg}} A : \tau; \Gamma'$, where Γ denotes the typing environment that is given as input in order to type-check expression A with type τ and Γ' denotes the environment entries that were not consumed in this derivation and can be used to prove further subgoals.

We note that every typing judgment of the form $\Gamma \vdash A : \tau$ is matched by an algorithmic counterpart of the form $\Gamma \vdash_{\mathsf{alg}} A : \tau; \Gamma'$, which is sound and complete. Intuitively, this means that if $\Gamma \vdash_{\mathsf{alg}} A : \tau; \Gamma'$ then $\Gamma \vdash A : \tau$ (*soundness*). Furthermore, if $\Gamma \vdash A : \tau$ then there exists Γ' such that $\Gamma \vdash_{\mathsf{alg}} A : \tau; \Gamma'$ (*completeness*).

We demonstrate the approach on the following algorithmic variants of the typing rules for variables (Var Alg) and pairs (⊗I Alg):

(Var Alg)
$$\frac{\Gamma \vdash \diamond \qquad x :!_k\phi \in \Gamma \qquad k \geq 1 \qquad \Gamma' = \Gamma\{x :!_{k-1}\phi / x :!_k\phi\}}{\Gamma \vdash_{\mathsf{alg}} x :!_1\phi; \Gamma'}$$

(⊗I Alg)
$$\frac{\Gamma \vdash_{\mathsf{alg}} M_1 : \tau_1; \Gamma' \qquad \Gamma' \vdash_{\mathsf{alg}} M_2 : \tau_2; \Gamma''}{\Gamma \vdash_{\mathsf{alg}} (M_1, M_2) :!_1(\tau_1 \otimes \tau_2); \Gamma''}$$

A variable x can be type-checked with type $!_1\phi$ under typing environment Γ if x is bound to core type ϕ with replication index $k \geq 1$ in Γ. In the returned environment Γ' that can be used to type-check further subgoals, x is instead bound to ϕ with reduced replication index $k - 1$.

The rule ⊗I Alg for pairs (M_1, M_2) exemplifies the treatment of environment splitting: the complete typing environment Γ is used to type-check M_1 with type τ_1. The resulting (possibly reduced) environment Γ' is then used to type-check M_2 with type τ_2, which upon success returns the final remaining environment Γ'' and allows us to type-check the pair (M_1, M_2) with type $!_1(\tau_1 \otimes \tau_2)$.

Example 4.4. *As an example, take $\Gamma = x :!_4\mathbb{Z}$, $A = (x, x)$, and $\tau =!_1(!_1\mathbb{Z}\otimes!_1\mathbb{Z})$. The algorithmic typing derivation to type-check A with type τ under Γ looks as follows:*

$$\frac{x :!_4\mathbb{Z} \vdash_{\mathsf{alg}} x :!_1\mathbb{Z}; x :!_3\mathbb{Z} \qquad x :!_3\mathbb{Z} \vdash_{\mathsf{alg}} x :!_1\mathbb{Z}; x :!_2\mathbb{Z}}{x :!_4\mathbb{Z} \vdash_{\mathsf{alg}} (x, x) :!_1(!_1\mathbb{Z}\otimes!_1\mathbb{Z}); x :!_2\mathbb{Z}}$$

We first type-check the left pair component x with type $!_1\mathbb{Z}$ under environment $x :!_4\mathbb{Z}$ and return the remaining environment $!_3\mathbb{Z}$, which is then used to type-check the right pair component x again with type $!_1\mathbb{Z}$. The pair is thus assigned type $!_1(!_1\mathbb{Z}\otimes!_1\mathbb{Z})$ and the remaining resources $x :!_2\mathbb{Z}$ are returned.

4.6.2 Base judgements and kinding

Both the well-formedness judgement and the kinding relation of our type system are already algorithmic. In particular, we will make use of the fact that deciding whether or not a type has kind `tnt` or `pub` is deterministic in the algorithmic version of the subtyping rules.

4.6.3 Subtyping

We introduce the rules of the algorithmic subtyping relation in Table 4.10. Since subtyping does not consume resources that are stored in the typing environment Γ, there is no need for a deterministic splitting of the typing environment. However, the algorithmic subtyping rules need to resolve the non-determinism of the original DF7 system that is due to the fact that the original subtyping rules are not syntax-driven. In fact, in DF7 there are (at most) four rules applicable to subtype $\Gamma \vdash \rho <: \tau$: (Sub Refl), (Sub Repl), (Sub Kind), and in the case that two types share the same top-level constructor one corresponding structural subtyping rule (e.g., (Sub Pair) in the case of pairs).

To remove the non-structural rule (Sub Repl) that encodes the fact that a resource can be used less often than its replication index, we hardcode it into all remaining rules that only operate on replication factor 1 in the original system. Additionally, we require all structural algorithmic subtyping rules to only operate on types $\tau =!_t\phi$ and $\rho =!_k\psi$, where $k \leq t$ and $\phi \neq \psi$, thus avoiding a clash with the reflexivity rule (Sub Refl Alg) that operates on types with the same core type.

The only remaining issue lies in resolving the non-determinism of the original (Sub Kind) rule, a rule that can be applied to show that a public type is a subtype of a tainted type. We resolve this non-determinism by letting the algorithmic kinding-based rule (Sub Kind Alg) only be applicable to two types ρ, τ that do not share the same top-level constructor, i.e. they are structurally different, which we denote by $\rho \neq_{\mathsf{str}} \tau$. Note that two types that share the same top-level constructor can still be subtyped using kinding by recursively applying the matching structural subtyping rules until one of the subgoals matches the premise of the rule (Sub Kind Alg).

This approach is both sound and complete for all but the case of subtyping two iso-recursive types. This is an artifact of our choice of not adapting the Amber rule but the iso-recursive subtyping proposed by Backes et al. [32, 33], which requires the recursive variable to occur only positively in the iso-recursive type (cf. (Sub Pos Rec) in Section 4.5.3).

For instance, given the above constraints, subtyping

$$\Gamma \vdash_{\mathsf{alg}} !_\infty(\mu\alpha.!_\infty(!_\infty\alpha \multimap !_\infty\mathsf{Unit})) <: !_\infty(\mu\alpha.!_\infty(!_\infty\alpha \multimap !_\infty(!_\infty\mathsf{Unit} \otimes !_\infty\mathsf{Unit})))$$

would not be possible, thus lacking kinding-based algorithmic subtyping for iso-recursive types. Therefore, our algorithmic type system contains two rules for subtyping two iso-recursive types: (Sub Kind Rec Alg), and (Sub Pos Rec Alg), respectively. The choice of which rule to apply is made deterministic by first checking whether the types are public and tainted.

4.6.4 Typing values and expressions.

The algorithmic typing rules for values are presented in Table 4.11, those for expressions in Table 4.12. The rules straightforwardly follow the intuition described in Section 4.6.1. We rely on typing annotation to guide the selection of appropriate rules and types, which we will explain in the following. With some abuse of notation, we let A range also over expressions with typing annotations.

In the case of sum and iso-recursive types the algorithmic system $\mathrm{DF7}_{\mathrm{alg}}$ relies on annotating the constructor as $h_\phi\ M$ to help with the choice of the appropriate missing type ϕ ($+/\mu$I Alg). Since in the typing rule ($\multimap$I) the type of the input is

(Sub Kind Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} \tau :: \mathsf{pub} \qquad \Gamma \vdash_{\mathsf{alg}} \rho :: \mathsf{tnt} \qquad \tau \neq_{\mathsf{str}} \rho}{\Gamma \vdash_{\mathsf{alg}} \tau <: \rho}$$

(Sub Refl Alg)

$$\frac{\Gamma \vdash \phi \qquad k \leq t}{\Gamma \vdash_{\mathsf{alg}} !_t\phi <: !_k\phi}$$

(Sub Sum Alg)

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{alg}} \tau <: \tau' \qquad \Gamma \vdash_{\mathsf{alg}} \rho <: \rho' \\ k \leq t \qquad \tau + \rho \neq \tau' + \rho'\end{array}}{\Gamma \vdash_{\mathsf{alg}} !_t(\tau + \rho) <: !_k(\tau' + \rho')}$$

(Sub Pair Alg)

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{alg}} \tau <: \tau' \qquad \Gamma \vdash_{\mathsf{alg}} \rho <: \rho' \\ k \leq t \qquad \tau \otimes \rho \neq \tau' \otimes \rho'\end{array}}{\Gamma \vdash_{\mathsf{alg}} !_t(\tau \otimes \rho) <: !_k(\tau' \otimes \rho')}$$

(Sub Fun Alg)

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{alg}} \tau' <: \tau \qquad \Gamma \vdash_{\mathsf{alg}} \rho <: \rho' \\ k \leq t \qquad \tau \multimap \rho \neq \tau' \multimap \rho'\end{array}}{\Gamma \vdash_{\mathsf{alg}} !_t(\tau \multimap \rho) <: !_k(\tau' \multimap \rho')}$$

(Sub Kind Rec Alg)

$$\frac{\begin{array}{c}\Gamma \vdash !_t(\mu\alpha.\tau) :: \mathsf{pub} \qquad \Gamma \vdash !_k(\mu\alpha.\rho) :: \mathsf{tnt} \\ \tau \neq \rho\end{array}}{\Gamma \vdash_{\mathsf{alg}} !_t(\mu\alpha.\tau) <: !_k(\mu\alpha.\rho)}$$

(Sub Pos Rec Alg)

$$\frac{\begin{array}{c}\Gamma, \alpha \vdash_{\mathsf{alg}} \tau <: \rho \qquad \alpha \text{ occurs only positively in } \tau \text{ and } \rho \qquad k \leq t \\ \neg(\Gamma \vdash !_t(\mu\alpha.\tau) :: \mathsf{pub} \wedge \Gamma \vdash !_k(\mu\alpha.\rho) :: \mathsf{tnt}) \qquad \tau \neq \rho\end{array}}{\Gamma \vdash_{\mathsf{alg}} !_t(\mu\alpha.\tau) <: !_k(\mu\alpha.\rho)}$$

Notation: Here $\tau \neq_{\mathsf{str}} \rho$ denotes that τ and ρ do not share the same top-level constructor (disregarding replication indices).

Table 4.10: Algorithmic subtyping relation (DF7$_{\text{alg}}$)

chosen non-deterministically, in the algorithmic variant we annotate the input as $\lambda x : \tau.A$ to guide type-checking ($\multimap$I Alg). To select an appropriate replication factor k, the rule (!I Alg) uses annotations of the form $M_{!k}$. We note that the typing of references already relies on typing annotations. The intuition behind the algorithmic typing of variables (Var Alg) and pairs ($\otimes$I Alg) was shown in Section 4.6.1. The typing of signature values does not consume any entries from the typing environment Γ and thus simply returns the complete environment untouched (Sig Alg).

The subtyping rule Sub is highly non-deterministic and can be applied at any time. We thus rely on manual annotations of the form $(A)_{\_<:\rho}$, describing the expected supertype ρ to only trigger the application of subtyping when explicitly necessary (Sub Alg). In the case of ($\otimes$E), (+E), and (μE) typing relies on the choice of an appropriate replication factor r of the value M. We eliminate the non-deterministic choice by requiring an explicit annotation of the form $M_{!r}$ ($\otimes$E Alg), (+E Alg), (μE Alg). In the latter two rules we make use of the environment operation $\mathsf{min}(\Gamma, \Gamma')$ to always select the lower replication factor of a variable binding that is contained in both of the environments Γ and Γ'. min does not affect type variables or their kinding. The cases of ($\multimap$E Alg), (Let Alg), and (Eq Alg) are rather straightforward: the environment is first used in the left premise and the remaining environment is then used in the right premise. Since the typing of references does not consume any entries from the environment

(Var Alg)

$$\frac{\Gamma \vdash \diamond \qquad x :!_k\phi \in \Gamma \qquad k \geq 1 \qquad \Gamma' = \Gamma\{x :!_{k-1}\phi / x :!_k\phi\}}{\Gamma \vdash_{\mathsf{alg}} x :!_1\phi; \Gamma'}$$

(Sig Alg)

$$\frac{M : \phi \in \Sigma \qquad M \in \{c, f\} \qquad \Gamma \vdash \diamond \qquad \emptyset \vdash \phi}{\Gamma \vdash_{\mathsf{alg}} M :!_1\phi; \Gamma}$$

(⊗I Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} M_1 : \tau_1; \Gamma' \qquad \Gamma' \vdash_{\mathsf{alg}} M_2 : \tau_2; \Gamma''}{\Gamma \vdash_{\mathsf{alg}} (M_1, M_2) :!_1(\tau_1 \otimes \tau_2); \Gamma''}$$

(+/μI Alg)

$$\frac{h : (\tau, \phi) \qquad \Gamma \vdash_{\mathsf{alg}} M : \tau; \Gamma' \qquad \Gamma \vdash \phi}{\Gamma \vdash_{\mathsf{alg}} h_\phi\ M :!_1\phi; \Gamma'}$$

($\multimap$I Alg)

$$\frac{\Gamma, x : \tau \vdash_{\mathsf{alg}} A : \rho; \Gamma'}{\Gamma \vdash_{\mathsf{alg}} \lambda x : \tau.A :!_1(\tau \multimap \rho); \Gamma' \backslash x}$$

(Read Alg)

$$\frac{\Gamma \vdash \tau \qquad \tau \notin \mathcal{OPP}}{\Gamma \vdash_{\mathsf{alg}} \mathsf{read}_{a:\tau} :!_1\mathsf{Read}\langle\tau\rangle; \Gamma}$$

(Read Opp Alg)

$$\frac{\Gamma \vdash \tau \qquad \tau \in \mathcal{OPP}}{\Gamma \vdash_{\mathsf{alg}} \mathsf{read}_{a:\tau} :!_1[\mathsf{Read}\langle\tau\rangle]_\infty; \Gamma}$$

(Write Alg)

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash_{\mathsf{alg}} \mathsf{write}_{a:\tau} :!_1\mathsf{Write}\langle\tau\rangle; \Gamma}$$

(!I Alg)

$$\frac{\Gamma \vdash_{\mathsf{alg}} M :!_1\phi; \Gamma' \qquad k > 0}{k\Gamma \vdash_{\mathsf{alg}} M_{!k} :!_k\phi; k\Gamma'}$$

Notation: We write $(\Gamma, x : \tau, \Gamma')\backslash x$ to denote Γ, Γ'.

Table 4.11: Algorithmic typing of values (DF7$_{\text{alg}}$)

(Sub Alg)
$$\frac{\Gamma \vdash_{\mathsf{alg}} A : \tau; \Gamma' \qquad \Gamma \vdash_{\mathsf{alg}} \tau <: \rho}{\Gamma \vdash_{\mathsf{alg}} (A)\_{<:\rho} : \rho; \Gamma'}$$

($\otimes$E Alg)
$$\frac{\Gamma \vdash_{\mathsf{alg}} M_{!r} :!_r(\tau_1 \otimes \tau_2); \Delta \qquad \Delta, x :!_r\tau_1, y :!_r\tau_2 \vdash_{\mathsf{alg}} A : \tau'; \Gamma'}{\Gamma \vdash_{\mathsf{alg}} \mathsf{let}\ (x, y) = M_{!r}\ \mathsf{in}\ A : \tau'; (\Gamma' \backslash x) \backslash y}$$

(+E Alg)
$$\frac{\Gamma \vdash_{\mathsf{alg}} M_{!r} :!_r(\tau_1 + \tau_2); \Delta \qquad \Delta, x :!_r\tau_1 \vdash_{\mathsf{alg}} A : \tau'; \Gamma' \qquad \Delta, x :!_r\tau_2 \vdash_{\mathsf{alg}} B : \tau'; \Gamma''}{\Gamma \vdash_{\mathsf{alg}} \mathsf{case}\ M_{!r}\ \mathsf{of}\ x\ \mathsf{in}\ A\ \mathsf{else}\ B\ : \tau'; \mathsf{min}(\Gamma', \Gamma'') \backslash x}$$

(μE Alg)
$$\frac{\Gamma \vdash_{\mathsf{alg}} M_{!r} :!_r\mu\alpha.\tau; \Delta \qquad \Delta, x :!_r\tau\{\mu\alpha.\tau/\alpha\} \vdash_{\mathsf{alg}} A : \tau'; \Gamma' \qquad \Delta \vdash_{\mathsf{alg}} B : \tau'; \Gamma''}{\Gamma \vdash_{\mathsf{alg}} \mathsf{unfold}\ M_{!r}\ \mathsf{as}\ x\ \mathsf{in}\ A\ \mathsf{else}\ B\ : \tau'; \mathsf{min}(\Gamma' \backslash x, \Gamma'')}$$

($\multimap$E Alg)
$$\frac{\Gamma \vdash_{\mathsf{alg}} M_1 :!_1(\tau \multimap \tau'); \Delta \qquad \Delta \vdash_{\mathsf{alg}} M_2 : \tau; \Gamma'}{\Gamma \vdash_{\mathsf{alg}} M_1 M_2 : \tau'; \Gamma'}$$

(Let Alg)
$$\frac{\Gamma \vdash_{\mathsf{alg}} A : \tau; \Delta \qquad \Delta, x : \tau \vdash_{\mathsf{alg}} B : \tau'; \Gamma'}{\Gamma \vdash_{\mathsf{alg}} \mathsf{let}\ x = A\ \mathsf{in}\ B : \tau'; \Gamma' \backslash x}$$

(Add-Noise Alg)
$$\frac{\Gamma \vdash_{\mathsf{alg}} M :!_1\mathbb{Z}; \Gamma'}{\Gamma \vdash_{\mathsf{alg}} \mathsf{add\_noise}^{s}_{\mathbb{Z}\to\mathbb{Z}}\ M :!_\infty\mathbb{Z}; \Gamma'}$$

(Ref Alg)
$$\frac{\Gamma \vdash \tau \qquad \tau \notin \mathcal{OPP}}{\Gamma \vdash_{\mathsf{alg}} \mathsf{ref}_\tau :!_\infty \mathsf{Ref}\langle\tau\rangle; \Gamma}$$

(Ref Opp Alg)
$$\frac{\Gamma \vdash \tau \qquad \tau \in \mathcal{OPP}}{\Gamma \vdash_{\mathsf{alg}} \mathsf{ref}_\tau : [!_\infty \mathsf{Ref}\langle\tau\rangle]_\infty; \Gamma}$$

(Eq Alg)
$$\frac{\Gamma \vdash_{\mathsf{alg}} M :!_\infty\tau; \Delta \qquad \Delta \vdash_{\mathsf{alg}} N :!_\infty\tau; \Gamma'}{\Gamma \vdash_{\mathsf{alg}} M = N :!_\infty \mathsf{Bool}; \Gamma'}$$

Notation: $\mathsf{min}((\Gamma, x :!_k\tau), (\Gamma', x :!_r\tau)) = \mathsf{min}(\Gamma, \Gamma'), x :!_{\mathsf{min}(k,r)}\tau$; $\mathsf{min}((\Gamma, \alpha), (\Gamma', \alpha)) = \mathsf{min}(\Gamma, \Gamma'), \alpha$; $\mathsf{min}((\Gamma, \alpha :: \kappa), ((\Gamma', \alpha :: \kappa)) = \mathsf{min}(\Gamma, \Gamma'), \alpha :: \kappa$; $\mathsf{min}(\emptyset, \emptyset) = \emptyset$. min is undefined in all other cases.

Table 4.12: Algorithmic typing of expressions (DF7$_{\mathrm{alg}}$)

Γ, the whole Γ is returned (Ref Alg) and (Ref Opp Alg). In the case of the easy inductive rule (Add-Noise), the returned environment inductively arises from the premise (Add-Noise Alg).

4.6.5 Formal results

Below we state the completeness and soundness result of the algorithmic variant DF7_{alg} of our type system.

We write $\langle A \rangle$ to denote the expression obtained by removing the typing annotations from A.

Theorem 4.2 (Completeness and Soundness of DF7_{alg}). *For every Γ, A, and τ, the following conditions hold:*

1. *If $\Gamma \vdash A : \tau$ then there exist Γ', A' such that $\Gamma \vdash_{\mathsf{alg}} A' : \tau; \Gamma'$ and $A = \langle A' \rangle$.*

2. *If $\Gamma \vdash_{\mathsf{alg}} A : \tau; \Gamma'$ then there exists Γ'' such that $\Gamma'' \vdash \langle A \rangle : \tau$ and $\Gamma = \Gamma' + \Gamma''$.*

4.7 A sealing-based cryptographic library

Originally, Morris [53] proposed the notion of dynamic sealing as a mechanism to protect program modules. While defining the semantics for a λ-calculus with dynamic sealing, Sumii and Pierce [54] later on observed a close correspondence with symmetric encryption. Bengtson et al. [31] showed how to encode a sealing-based cryptographic library for RCF using pairs, functions, references, and lists. In the following we propose a sealing-based cryptographic library for randomized symmetric cryptography that is well-typed in our type system.

4.7.1 Standard sealing-based libraries

The core idea of using seals to model cryptography is the following: a global reference is used to store a list of message-ciphertext pairs. This reference can only be accessed via the sealing and unsealing functions. The plaintext that is to be encrypted is paired with a fresh value, which represents the ciphertext. This message-ciphertext pair is added to the list (sealing). To decrypt a ciphertext, the latter is looked up in the list. If the ciphertext is in the list, then the corresponding message is returned (unsealing).

The symmetric key consists of the pair of the sealing and unsealing function. As shown by Bengtson et al. [31], we can model public-key cryptography (and other primitives) in a similar way: the sealing function corresponds to the encryption key, the unsealing function corresponds to the decryption key.

4.7.2 Affine sealing-based library

As we have shown in Section 4.4, the crucial ingredient to enforce differential privacy is the affine usage of resources. In a distributed setting, one has to make sure that the sensitive (affine) data that is exchanged over the network does not get duplicated and processed more than once. Otherwise a single entry in the (distributed) database could have a huge, and not statically predictable, impact on the final result. Since we cannot prevent the opponent from duplicating public ciphertexts, we must enforce that the content of each ciphertext cannot be processed more than once.

This can be achieved by encoding the sealing mechanism using a *affine reference* to store the list of message-ciphertext pairs. The resulting sealing function behaves as in the non-affine setting. In the unsealing function, the list is automatically removed from the referenced memory after being read. The list is then searched for the message-ciphertext pair. If the pair is contained in the list, the function will return the message and store the list pruned of the pair back into the reference. Thus, further decryptions of the same ciphertext are no longer possible. If the pair is not found, the complete list is stored back into the reference[4]. We will present the full implementation of our affine cryptographic library in the next section.

4.7.3 Implementation of the cryptographic library

Below, we present the details of our cryptographic library. We first show the typed interface of the cryptographic library functions and then their implementation.

Typed interface. We now list a typed interface of the most important functions in our cryptographic library. Except for the use of affine references and the possibility to re-store message-ciphertext pairs into the seal reference, the code of the functions is mostly standard. Note that all functions can be typed under the empty environment and can thus be given an arbitrary replication index.

Sealing and Unsealing. The type of the references we use for seals is defined as $\mathsf{SealRef}\langle\tau\rangle \triangleq \mathsf{Ref}\langle\mathsf{List}\langle\tau\otimes !_\infty\mathbb{Z}\rangle\rangle$. The sealing and unsealing functions are both initialized with the seal reference. The sealing function can then be used to seal a value of type τ, the unsealing function to unseal a public value of type $!_\infty\mathbb{Z}$.

$$\begin{aligned}&\mathit{seal} : !_\infty\mathsf{SealRef}\langle\tau\rangle \multimap !_\infty((\tau + (\tau\otimes !_\infty\mathbb{Z})) \multimap !_\infty\mathbb{Z})\\ &\mathit{unseal} : !_\infty\mathsf{SealRef}\langle\tau\rangle \multimap !_\infty(!_\infty\mathbb{Z} \multimap \mathsf{Option}\langle\tau\rangle)\end{aligned}$$

A seal is a pair of the initialized sealing and unsealing functions $\mathsf{Seal}\langle\tau\rangle \triangleq !_\infty((\tau + (\tau\otimes !_\infty\mathbb{Z})) \multimap !_\infty\mathbb{Z})\otimes !_\infty(!_\infty\mathbb{Z} \multimap \mathsf{Option}\langle\tau\rangle)$. To create a seal the following function *mkSeal* is available:

$$\mathit{mkSeal} : !_\infty\mathsf{Unit} \multimap !_\infty\mathsf{Seal}\langle\tau\rangle$$

Key Generation and Enc-/ Decryption. A symmetric key is a seal: it can be used for encrypting (by calling the sealing function) and for decrypting (using the unsealing function). We define $\mathsf{SKey}\langle\tau\rangle \triangleq \mathsf{Seal}\langle\tau\rangle$ and provide the following typed interface for the functions for key generation, randomized symmetric encryption, and symmetric decryption:

$$\begin{aligned}&\mathit{mkSKey} : !_\infty\mathsf{Unit} \multimap !_\infty\mathsf{SKey}\langle\tau\rangle\\ &\mathit{senc} : \mathsf{SKey}\langle\tau\rangle \multimap (\tau \multimap !_\infty\mathbb{Z})\\ &\mathit{sdec} : \mathsf{SKey}\langle\tau\rangle \multimap (!_\infty\mathbb{Z} \multimap \mathsf{Option}\langle\tau\rangle)\end{aligned}$$

[4] Note that in the case of non-affine payloads the decrypt-once restriction is not necessary. The sealing mechanism for such values could thus be encoded using a standard non-destructive reference.

Implementation details. A lot of the functions we define on lists will be recursive. In [36] the authors describe how a fixpoint combinator can be simulated in a standard way. They also explain how an explicit fixpoint operator can be added to the type system. To aid the legibility of our code we thus add the following typing rule

$$\textsc{Fix}\quad \frac{\Gamma, f :!_\infty(\tau \multimap \tau') \vdash A : \tau \multimap \tau'}{\infty\Gamma \vdash \mathsf{fix}\ f.A : \tau \multimap \tau'}$$

Convention: Here and in the following we write true to denote $\mathsf{inl}\ ()$ as well as false to denote $\mathsf{inr}\ ()$. Furthermore, we use the notations $\mathsf{none} \triangleq \mathsf{inl}\ ()$ and $\mathsf{some}\ x \triangleq \mathsf{inr}\ x$ and $\mathsf{nil} \triangleq \mathsf{fold}\ \mathsf{inl}\ ()$ as well as $\mathsf{cons}\ h\ t \triangleq \mathsf{fold}\ \mathsf{inr}\ (h, t)$
To concatenate two lists we can use the following function:

```
$concat$ :
$!_1\mathsf{List}\langle\tau\rangle \multimap !_1(!_1\mathsf{List}\langle\tau\rangle \multimap !_1\mathsf{List}\langle\tau\rangle)$


    let $concat$ =
    fix $g.\lambda l_1.\lambda l_2.$
    unfold $l_1$ as $l_1'$ in
          case $l_1'$ of $l_1''$ in $l_2$
          else let $(h, t) = l_1''$ in
              let $r = g\ t\ l_2$ in
              cons $h\ r$;
```

Lookup Function. Given a list of pairs with exponential second component and a handle of the same exponential type, this function will be used to look up the first occurrence of that handle as the second component of an entry in the list and return the pair containing the first component of that entry in the list and the list from which the entry has been removed.

```
$lookup_R$ :
     $!_\infty\phi \multimap !_1(!_1\mathsf{List}\langle\tau\otimes!_\infty\phi\rangle \multimap$
     $!_1(\mathsf{List}\langle\tau\otimes!_\infty\phi\rangle \multimap$
     $!_1(!_1\mathsf{Option}\langle\tau\rangle\otimes!_1\mathsf{List}\langle\tau\otimes!_\infty\phi\rangle)))$

let $lookup_R$ =
     fix $g.\lambda c.\lambda o.\lambda l.$
     unfold $l$ as $l'$ in
         case $l'$ of $l''$ in (none, $o$)
         else let $(h, t) = l''$ in
             let $(h_1, h_2) = h$ in
             let $comp = (c = h_2)$ in
             case $comp$ of $x$ in
                 (some $h_1$, $concat(o, t)$)
             else $g\ \ c$ (cons $(h_1, h_2)\ o$) $t$;
```

Seals. The sealing function is defined as follows:

```
let $seal$ =
$\lambda s.\lambda m.$
let $(r, w) = s$ in
let $state = r()$ in
case $m$ of $m'$ in
        let $c = create\_new()$ in
        let $p = (m', c)$ in
        case $state$ of $l$ in
            let $upd = w$ (some (cons $p$ nil)) in
            $c$
        else
            let $upd = w$ (some (cons  $p$ $l$)) in
            $c$
else
        let $(m_o, c_o) = m'$ in
        let $p = (m_o, c_o)$ in
        case $state$ of $l$ in
            let $upd = w$ (some (cons $p$ nil)) in
            $c_o$
        else
            let $upd = w$ (some (cons  $p'$ $l$)) in
            $c_o$ ;
```

Here the function $create\_new$:$!_\infty$Unit $\multimap !_\infty\mathbb{Z}$ creates a fresh public value. This can be encoded by creating a fresh reference (ref$_{!_\infty\mathbb{Z}}$) and then using subtyping to give the reference type $!_\infty\mathbb{Z}$.

The unsealing function is defined as follows:

```
let $unseal$ =
$\lambda s.\lambda c.$
let $(r, w) = s$ in
let $state = r()$ in
case $state$ of $l$ in none
else
        let $res = lookup_R$ $c$ nil $l$ in
        let $(m, l') = res$ in
        let $upd = w$ (some $l'$) in
        $m$;
```

The following function can be used to create a seal:

```
let $mkSeal$ =
$\lambda\_.$
let $(r, w)$ = ref in
$(seal\ (r, w), unseal\ (r, w))$;
```

Cryptographic Operations. Symmetric key generation is defined as follows:

```
let $mkSKey$ =
$\lambda\_.$
let $(e, d) = mkSeal()$ in
$(e, d)$;
```

Symmetric encryption is defined as:

```
let $senc$ =
$\lambda k.\lambda m.$
let $(e, d) = k$ in
$e$ (inl $m$);
```

The corresponding decryption function looks as follows:

```
let $sdec$ =
$\lambda k.\lambda c.$
let $(e, d) = k$ in $d\ c$;
```

Public key cryptography. As shown by Bengtson et al. [31], we can model randomized public-key cryptography (as well as other primitives) in a similar way: the sealing function corresponds to the encryption key, while the unsealing function corresponds to the decryption key. For instance, the type of an encryption key for a secret number of type $!_1\mathbb{Z}$ is $!_\infty(!_1\mathbb{Z} \multimap !_\infty\mathbb{Z})$. The kinding relation guarantees that this type is public and, thus, that the encryption key can be made available to the opponent, who can later use it by giving it the supertype $!_\infty(!_\infty\mathbb{Z} \multimap !_\infty\mathbb{Z})$. Honest parties will give decrypted messages type $!_1\mathbb{Z}$, which is indeed a supertype of $!_\infty\mathbb{Z}$. Indeed, the type $!_1\mathbb{Z}$ is tainted, that is, it does not provide any authenticity guarantee.

Formally, asymmetric public key encryption is defined as follows:

```
$\mathsf{DKey}\langle\tau\rangle \triangleq \mathsf{Seal}\langle\tau\rangle$

$\mathsf{EKey}\langle\tau\rangle \triangleq (\tau + (\tau\otimes !_\infty\mathbb{Z})) \multimap !_\infty\mathbb{Z}$

$\mathit{mkDKey}$ :$!_\infty\mathsf{Unit} \multimap !_\infty\mathsf{DKey}\langle\tau\rangle$

let $\mathit{mkDKey} =$
$\lambda\_.$
let $(e, d) = \mathit{mkSeal}()$ in
$(e, d)$;

$\mathit{mkEKey}$ :$!_\infty\mathsf{DKey}\langle\tau\rangle \multimap !_\infty\mathsf{EKey}\langle\tau\rangle$

let $\mathit{mkEKey} =$
$\lambda dk.$
let $(e, d) = dk$ in
$e$;

$\mathit{aenc}$ :$!_1\mathsf{EKey}\langle\tau\rangle \multimap !_1(\tau \multimap !_\infty\mathbb{Z})$

let $\mathit{aenc} =$
$\lambda ek.\lambda m.$
$ek$ (inl $m$);

$\mathit{adec}$ :$!_1\mathsf{DKey}\langle\tau\rangle \multimap !_1(!_\infty\mathbb{Z} \multimap !_1\mathsf{Option}\langle\tau\rangle)$

let $\mathit{adec} =$
$\lambda dk.\lambda c.$
let $(e, d) = dk$ in $d\ c$;
```

4.7.4 Symbolic soundness of cryptographic library

Backes et al. [55] proved the computational soundness of a standard sealing-based library, establishing a semantic link between sealing-based cryptographic libraries and traditional Dolev-Yao constructor-based libraries. Here we show that our affine sealing-based cryptographic library can be proven sound with respect to a standard sealing-based library.

Intuitively, we have to make sure that the content of each ciphertext is not processed more than once, that is, the protocol is secure against replay attacks. There exist many standard techniques to defend against replay attacks, e.g., nonce-handshakes or session keys. We generalize this concept to the notion of *guarded decryption*. Intuitively, we let the cryptographic library contain multiple guarded decryption functions $\mathit{dec\&check}_i$, which replace the original decryption functions. A guarded decryption function takes as input the decryption key and the ciphertext as well as an additional *guard*. It then unseals the ciphertext, applies a boolean check to the content of the ciphertext and the guard and only returns the plaintext if the check succeeded (otherwise, the plaintext is stored back into the seal). Below we demonstrate the concept of guarded decryption on a concrete example. We will introduce another guarded decryption function in the case study (cf. Section 4.8).

Guarded decryption (example). For instance, nonce-handshakes can be encoded in terms of a corresponding guarded decryption function $dec\&check_{nonce}$ that takes the nonce as a guard and performs an equality check between the guard and the nonce that is contained in the ciphertext alongside the payload. Here, $dec\&check_{nonce}$ is of the following type:

$$\begin{aligned}&!_1\mathsf{DKey}\langle !_\infty\mathsf{Nonce} \otimes \tau\rangle \multimap !_1(!_\infty\mathbb{Z} \multimap \\ &!_1(!_\infty\mathsf{Nonce} \multimap !_1\mathsf{Option}\langle\tau\rangle))\end{aligned}$$

The function is implemented as follows:

```
let $dec\&check_{nonce}$ =
  $\lambda k.\lambda c.\lambda g.$
  let $r = adec\ k\ c$ in
  case $r$ of $r'$ in none
  else
      let $(n, m) = r'$ in
      case $(n = g)$ of $z$ in some $m$
      else
          let $(e, d) = k$ in
          let $restore = e$ (inr $(m, c)$) in
          none;
```

Symbolic soundness In order to show the soundness of the affine sealing-based library, we have to put a restriction on the guarded decryption functions and on the usage of the guards in the protocol code. Intuitively, we say that a cryptographic library is *valid* if for each ciphertext c and key k, there exists at most one i and one guard g such that $dec\&check_i\ k\ c\ g$ succeeds (*guard uniqueness*). We also say that a program P is $\mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}$-valid if for every opponent O and every i, $O(\mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}; P)$ never calls $dec\&check_i$ more than once with the same guard (*guard usage affinity*). We give some insights on how to check guard uniqueness and guard usage affinity at the end of this chapter.

We now give a formal definition of guard uniqueness and guard usage affinity. The definitions rely on the notion of *well-formed keys*. A key is well-formed if the list of message-ciphertext pairs in the corresponding seal does not contain the same ciphertext twice.[5]

Definition 4.6 (Guard Uniqueness). *A cryptographic library $\mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}$ is* valid *if for each ciphertext c, each well-formed key k, and each guard g there exists a*

[5] In our cryptographic library all keys are well-formed by construction.

function $dec\&check_i \in \mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}$ *and a value* M *such that for all* N *and all functions* $dec\&check_j \in \mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}$ *it holds that if*

$$dec\&check_j\ c\ k\ g \rightarrow^* \mathsf{some}\ N$$

then $N = M$ *and* $i = j$*. We say that each guard is* unique *for* $\mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}$.

Definition 4.7 (Guard Usage Affinity)**.** *A program* P *is* $\mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}$*-valid for a cryptographic library* $\mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}$ *if for every opponent* O*, every guard* g*, and every function* $dec\&check_i \in \mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}$ *it holds that whenever there exist an evaluation context* C*, a ciphertext* c*, and a key* k *such that*

$$O(\mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}; P) \rightarrow^* C[dec\&check_i\ c\ k\ g] \rightarrow^* C[\mathsf{some}\ N]$$

for some value N *then there does not exist an evaluation context* C' *such that*

$$C[\mathsf{some}\ N] \rightarrow^* C'[dec\&check_i\ c'\ k'\ g]$$

for some key k' *and ciphertext* c'*.*

In the following, we write $\mathcal{L}_{\mathsf{crypto}}$ to denote the standard sealing-based library, which is essentially obtained from $\mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}$ by replacing destructive references with non-destructive references. We can finally state the soundness result for our affine sealing-based cryptographic library.

Theorem 4.3 (Symbolic soundness of the cryptographic library)**.** *Let* $\mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}$ *be a valid cryptographic library and* P *be a* $\mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}$*-valid program. For all opponents* O *and values* M*,* $O(\mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}; P) \rightarrow^*_p M$ *if and only if* $O(\mathcal{L}_{\mathsf{crypto}}; P) \rightarrow^*_p M$*.*

Proof Sketch. The result can be proven by observing that $\mathcal{L}_{\mathsf{crypto}}$ and $\mathcal{L}^{\mathsf{lin}}_{\mathsf{crypto}}$ behave in the same manner as long as the $dec\&check_i$ functions in $\mathcal{L}_{\mathsf{crypto}}$ do not successfully decrypt the same ciphertext more than once. This property is enforced by guard uniqueness and guard usage affinity.. □

Checking guard uniqueness. Guard uniqueness is a semantic property on the cryptographic library that is most often trivial to check. For instance, in the case of the guarded decryption function $dec\&check_{nonce}$, which checks whether the nonce N that is contained in the ciphertext alongside the payload corresponds to the guard nonce N_g, it is clear that there exists only one guard N_g, for which this check will succeed, namely N.

Checking guard usage affinity. The affine usage of guards can be checked using existing techniques, e.g., AF7, the type system presented in Chapter 2 for enforcing affine authorization policies in $\mathrm{RCF}_{\mathrm{AF7}}$. Since $\mathrm{RCF}_{\mathrm{DF7}}$ and $\mathrm{RCF}_{\mathrm{AF7}}$ constitute only slightly different variants of RCF, AF7 can be applied to our protocol code by considering the $\mathsf{add\_noise}^{s}_{\mathbb{Z}\to\mathbb{Z}}$ primitive as a library function typed with $\mathsf{Un} \rightarrow \mathsf{Un}$. The other primitives can be defined by encoding. We exemplify how to check that the guard is used only once per guarded decryption on the function $dec\&check_{nonce}$. In AF7 this decryption function is type-checked with type

$$\begin{array}{l} k : \mathsf{DKey}\langle \mathsf{Un} * \tau^{\mathrm{AF7}} \rangle \to c : \mathsf{Un} \to \\ g : \{x : \mathsf{Un} \mid \mathit{FreshGuard}_i(x)\} \to \mathsf{Option}\langle \tau^{\mathrm{AF7}} \rangle \end{array}$$

where the predicate $\mathit{FreshGuard}_i(x)$ may be assumed at most *once* per nonce x. Intuitively, this means that the code will only type-check in AF7 if each guarded decryption function is called at most once per guard, thus enforcing the affine guard usage.

Although we could have embedded one of these techniques in our type system, we intentionally factored out the problem of preventing replay attacks to keep the presentation clean and simple and to focus on the novel concepts introduced in this chapter.

4.8 Case study

In this section we demonstrate the usefulness of our approach by analyzing a recently proposed protocol for non-tracking web analytics (*NTWA*) by Akkus et al. [11]. The system promises both differential privacy guarantees to the clients visiting a webpage and good quality analytics to the publishers of a webpage. It requires no additional authority that is not present in current web analytics scenarios.

Although the authors manually prove that the deployed noise mechanism that protects aggregated user data is indeed differentially private and they show how the system prevents different attacks, there is no formal proof that the overall system provides distributed differential privacy guarantees to the clients. In fact, it is not clear in general how to transfer the privacy results of the noise mechanism to establish privacy results for the overall protocol.

In the following, we give a brief description of the *NTWA* system and show the results of our analysis that include a newly discovered attack, a fix for the protocol, and the verified differential privacy result for the revised system.

4.8.1 System overview

The *NTWA* system comprises the three entities that can be found in standard web analytics scenarios: clients, publishers, and data aggregators. Intuitively, the provider of a website (*publisher*) uses a third party web analytics service (*data aggregator*) to gain aggregated information about the users visiting the site (*clients*). In today's systems, the aggregator collects user information by tracking the client's behavior across the web and constructing a user profile, which poses a threat to the client's privacy. The *NTWA* system follows a different approach by assuming the clients voluntarily answer queries about their personal data in exchange for better privacy guarantees.

The query mechanism of the *NTWA* system is depicted in Figure 4.1 (cf. Section 4.2). Intuitively, the client stores its personal information in a local database (e.g., demographics and browsing history). When visiting a webpage (step 1 in Figure 4.1), a client will be asked to answer some queries (step 2). These queries are stored at a fixed URL on the webpage. They may come from either the publisher or the aggregator and can be rather complex (e.g., SQL) but only allow for 'yes' or 'no' answers, e.g., "*Are you between 35 and 50 years old?*" The client encrypts its answers to these queries with the public key of the data aggregator (step 3) before sending them to the publisher, that acts as a proxy (step 4). The publisher collects answers from multiple clients and creates some fake answers (noise), encrypts them, and mixes the real and fake answers (step 5), which it then forwards to the aggregator (step 6). The data aggregator decrypts all the received query results, adds them together, adds some own noise to the results (step 7), and sends the signed "double noisy" results back to the publisher (step 8). The publisher subtracts its own noise from the received analytics to obtain its final aggregated result (step 9).

Overall, both the publisher and the data aggregator obtain analytics about the clients that visited the publisher's webpage, but both results are not exact: the publisher cannot remove the noise that the aggregator added and vice-versa. As the authors show, the noise addition mechanisms they employ (variations of a discretized Laplace noise addition) are both differentially private, thus, intuitively, one should not be able to draw conclusions about a client's personal database given the aggregated and noisy query results.

The trust assumptions underlying the *NTWA* protocol are as follows: (*i*) The client is trusted; (*ii*) the publishers can be selfishly malicious; (*iii*) the aggregator is honest-but-curious (HbC); (*iv*) the aggregator and the publisher do not collude, which is crucial to prevent the removal of the "double noise".

We finally mention that the complete *NTWA* system also includes an auditing mechanism that is used to detect malicious publisher behavior, like dropping client answers. This mechanism is not necessary to achieve differential privacy and thus we omit it from our analysis, although it is useful in practice to detect attacks and identify malicious parties. For a more comprehensive overview of the system we refer to the original paper [11].

4.8.2 Attacking and Fixing the protocol

As we have shown in Section 4.7, the soundness of our approach relies on the affine usage of each ciphertext content. However, the *NTWA* protocol does not impose any restriction on the decryption of the encrypted client answers performed by the aggregator, thus allowing the same ciphertext to be decrypted multiple times, which leads to the replay attack described in Section 4.2.

The fix. We propose the following minor modification to the protocol to prevent the snapshot attack. As we have seen, the problem is that one answer of a client may influence the final tally multiple times. We note that each query is associated with a query ID and a query end time after which answers to that query will no longer be processed. To ensure that each encrypted client answer is processed only once, we require the aggregator to perform the following steps: For each query identifier the aggregator stores the encrypted answers it received in response to that query in a *duplicate-free list*. At the query end time, all ciphertexts associated to a query will be decrypted and the aggregator will check that the query identifier inside the client's answer corresponds to the expected one. If so, it will add the client's answer to the tally and proceed as expected, if not, it will discard the answer. In other words, the decryption is guarded and the guard consists of the query identifier paired with the ciphertext. Note that this requires a public encryption scheme that does not allow for re-encryption. After the query end time, the stored ciphertexts will be discarded.

4.8.3 Analysis of the revised protocol

The analysis of the protocol has to take two different threat-levels into account: the protocol has to be secure both against a network-level attacker and against compromised parties, such as a malicious publisher or an HbC data aggregator.

As usual in protocol analysis, the fact that an opponent has control over the network can be modeled by writing the protocol participants as cascaded functions that can be called and scheduled by the opponent at will. These functions take messages sent over the network as arguments when called by the opponent and return the messages they would otherwise output on the network. The potential compromise of the non-collaborating publisher and aggregator can be modeled by implementing *two* versions of the protocol: The former models the attack scenario in which the publisher is assumed to be honest and the aggregator to be HbC, the latter assumes an honest aggregator and a selfishly malicious publisher. In the former, the aggregator follows the protocol but leaks all data to the attacker, while in the latter the publisher role is directly played by the attacker.

While it is easy to verify the protocol under the assumption that all participants are honest and differential privacy has to be proven only with respect to a network-level opponent, verifying the protocol for an honest publisher and HbC data aggregator is currently out of the scope of our analysis technique. This is due to the fact that the noise mechanism employed by the publisher (a discretized version of Laplace noise with resampling) is a mechanism that gives only a weaker form of ϵ, δ-approximate differential privacy. Extending our type system to enforce approximate differential privacy is an interesting direction for future work.

We thus focus on the attack scenario in which the publisher is considered malicious and the data aggregator is honest. When analyzing the original *NTWA* protocol under this trust assumption we discovered the snapshot attack. As we will show in the next section, we can model the fixed protocol using a guarded decryption function $dec\&check_{NTWA}$ for the encrypted client answers. The decryption function takes the pair of the ciphertext and the associated query identifier as guard and checks that the query identifier corresponds to the one given in the client's answer. We successfully type-checked the fixed protocol, showing

ϵ-DDP against a malicious publisher and a network level opponent. The exact details of this analysis are presented in the next section.

4.8.4 Code of the analysis

In this section we provide our model of the fixed *NTWA* protocol in which we assume the aggregator to be fully trusted, but the publisher to be selfishly malicious (under the complete control of the opponent). As described above, we model the fact that the opponent has control over the network by writing the protocol participants as cascaded functions to be called and scheduled by the opponent at will. We model both the honest client for whose data we want to guarantee ϵ-differential privacy and the honest aggregator.

First, we show the signature functions and types that are used in the model. We then explain some necessary auxiliary functions before showing the guarded decryption function $dec\&check_{NTWA}$. We conclude by showing the implementations of the honest protocol participants.

Note that the *NTWA* protocol uses Laplace noise addition to provide noisy bucket counts. Since these counts are integers, the protocol applies a rounding function (floor) to the drawn Laplace noise in order to operate on integers. Since the discrete Laplace distribution is the discrete counterpart of the continuous Laplace distribution, we directly use our discrete $\mathsf{add\_noise}^{s}_{\mathbb{Z}\to\mathbb{Z}}$ operation to model the combination of real-valued Laplace noise and rounding.

Signature functions and types used in the model. In order to model the protocol we assume our signature Σ to contain the types and functions explained in the following. We note that any function contained in the signature must be 1-sensitive.

We will use the type $\mathbb{Z}$ to model (discrete) buckets and counts. Note that while we use integers, these counts will in fact be natural numbers, i.e., we will never encounter negative counts. Furthermore, we make use of type $\mathsf{Set}\langle\phi\rangle$ describing multisets of elements of (core) type ϕ and assume them to be also contained in signature Σ. Sets are used to model databases and also prove helpful in counting buckets as we will describe below. The distance on sets $\delta_{\mathsf{Set}\langle\phi\rangle}$ is the symmetric difference (the number of entries that are contained in one but not in the other set.

We now focus on some helpful functions that we include in the signature. For integers it can easily be seen that the following conversion function has sensitivity 1 and can thus be added to Σ:

$$\mathbb{Z}2set : \; !_{2k}\mathbb{Z} \multimap !_{k}\mathsf{Set}\langle\mathbb{Z}\rangle \qquad k \in \mathbb{R}^{\geq 0} \cup \{\infty\}$$

This function can be used to create the singleton set $\{z\}$ given an integer z as input. Note that while the two numbers 4 and 5 have a distance of 1, the distance of the set $\{4\}$ and $\{5\}$ is 2, thus requiring the argument of $\mathbb{Z}2set$ to have replication index 2 to guarantee 1-sensitivity.

In order to model the protocol, we furthermore assume the following 1-sensitive functions to be contained in the signature (we always assume $k \in \mathbb{R}^{\geq 0} \cup \{\infty\}$):

$$\begin{array}{l}
<, >, \leq, \geq\colon\ (!_\infty \mathbb{Z} \otimes !_\infty \mathbb{Z}) \multimap !_\infty \mathsf{Bool} \\
+, - :\ (!_k \mathbb{Z} \otimes !_k \mathbb{Z}) \multimap !_k \mathbb{Z} \\
\cup, \cap, \setminus :\ (!_k \mathsf{Set}\langle \phi \rangle \otimes !_k \mathsf{Set}\langle \phi \rangle) \multimap !_k \mathsf{Set}\langle \phi \rangle \\
\mathit{setfilter} :\ (!_\infty (!_\infty \phi \multimap !_\infty \mathsf{Bool}) \otimes !_k \mathsf{Set}\langle \phi \rangle) \multimap !_k \mathsf{Set}\langle \phi \rangle \\
\mathit{setmap} :\ (!_\infty (\phi_1 \multimap \phi_2) \otimes !_k \mathsf{Set}\langle \phi_1 \rangle) \multimap !_k \mathsf{Set}\langle \phi_2 \rangle
\end{array}$$

The mathematical operations and inequalities and the set operations behave as expected. The function $\mathit{setmap}(f, S)$ applies the function f to all elements of the set S. The function $\mathit{setfilter}(f, S)$ returns the set of all elements in S for which the function f returns true.

All of the above functions (or slightly modified versions) have been shown to be 1-sensitive by Reed and Pierce [36] (either by hand or by encoding and type-checking them using their type system).

We furthermore can implement the following functions (see [36] or slight modifications thereof).

$$\begin{array}{l}
\mathit{setsplit} :\ (!_\infty (!_\infty \phi \multimap !_\infty \mathsf{Bool}) \otimes !_k \mathsf{Set}\langle \phi \rangle) \multimap \\
\qquad\qquad (!_k \mathsf{Set}\langle \phi \rangle \otimes !_k \mathsf{Set}\langle \phi \rangle) \\
\mathit{sum} :\ (!_\infty (!_\infty b \multimap !_k \mathbb{Z}) \otimes !_k \mathsf{Set}\langle b \rangle) \multimap !_k \mathbb{Z} \\
\mathit{size} :\ !_k \mathsf{Set}\langle \phi \rangle \multimap !_k \mathbb{Z} \\
\mathit{map} :\ !_\infty (\tau \multimap \rho) \multimap (\mathsf{List}\langle \tau \rangle \multimap \mathsf{List}\langle \rho \rangle) \\
\mathit{length} :\ !_k \mathsf{List}\langle \tau \rangle \multimap (!_k \mathsf{List}\langle \tau \rangle \otimes !_\infty \mathbb{Z})
\end{array}$$

The function *setsplit* takes a boolean function f and a set S and returns two sets: one containing all the elements of S for which the function f returned true, the other containing all the remaining elements of S. The function *size* S returns the size of the set S as an integer. The *size* function is a special case of the function *sum* (f, S) which returns $\sum_{s \in S} \mathit{clip}(f\ s)$, where *clip* clips an integer number to an integer in $[-1, 1]$. The clipping is crucial to guarantee that *sum* is 1-sensitive. A function of sensitivity 1 preserves the distance between inputs. Consider two sets of type $\mathsf{Set}\langle \mathbb{Z} \rangle$: $S_1 = \{1\}$ and $S_2 = \{101\}$. By our definition of distance, the two sets have a (symmetric) distance of 2, but their elements have a distance of 100. If the *sum* function did not use clipping, the result of applying *sum* $(\lambda x.x)$ to the two databases would have a distance of 100. The function *map* applies a function to all the elements of a list, the function *length* returns the pair of the list itself and its length.

Using the above functions we can model a histogram function that takes a set s of (non-negative) integers and an upper bound $\mathcal{B}$ and returns a list of integers with $\mathcal{B}$ elements. The i-th entry in the list represents how often the number i occurs in s.

Buckets are encoded using natural numbers, which we represent by integers. We assume $\mathcal{B} \in \mathbb{Z}$ buckets, where $\mathcal{B} \geq 0$, and one bucket N/A for query results for which no other bucket is applicable. We write 0 to denote the N/A-bucket and $i \in \mathbb{Z}$ to denote the i-th bucket, where $i \in \{1, \ldots, \mathcal{B}\}$. Integers $z > \mathcal{B}$ or $z < 0$ do not describe a valid bucket.

We write $\mathsf{Q_{Id}} \triangleq !_\infty\mathbb{Z}$ to denote the query ID of the query that the selected bucket was given as answer to.

We let $\mathsf{Un} \in \mathcal{OPP}$ denote an arbitrary opponent type, e.g., $!_\infty\mathbb{Z}$.

Guarded decryption. Our implementation of the modified protocol uses a guarded decryption function $\mathit{dec\&check}_{NTWA}$ that is encoded below.
Here, $\mathit{dec\&check}_{NTWA}$ has the following type:

$$!_1\mathsf{DKey}\langle \mathsf{Q_{Id}} \otimes !_1\mathbb{Z}\rangle \multimap !_1(!_\infty\mathbb{Z} \multimap \\ !_1(!_1(!_\infty\mathbb{Z} \otimes \mathsf{Q_{Id}}) \multimap !_1\mathsf{Option}\langle !_1\mathbb{Z}\rangle))$$

The function is implemented as follows:

```
let $\mathit{dec\&check}_{NTWA}$ =
  $\lambda k.\lambda c.\lambda g.$
  let $(g_c, g_q) = g$ in
  case $(c = g_c)$ of $\mathit{check}_c$ in
      let $r = \mathit{adec}\ k\ c$ in
      case $r$ of $r'$ in none
      else
          let $(q, m) = r'$ in
          case $(q = g_q)$ of $z$ in some $m$
          else none
              let $(e, d) = k$ in
              let $\mathit{restore} = e$ (inr $(m, c)$) in
              none
  else none;
```

It takes a pair of a ciphertext and the current query identifier as guard and checks whether it is decrypting that particular ciphertext and whether the query identifier that was encrypted by the client corresponds to the one of the guard. We can prove uniqueness and affinity of each guard pair, thus showing that each ciphertext will only be decrypted once per correct query identifier.

Auxiliary functions. We present some of the auxiliary functions that are necessary for implementing the protocol participants. Using the above mentioned signature functions we can model a histogram function that takes a set s of non-negative integers and an upper bound $\mathcal{B}$ and returns a list of integers with $\mathcal{B}$ elements. The i-th entry in the list represents how often the number i occurs in s.

```
$\mathit{hist}' :\ !_\infty\mathbb{Z} \multimap (!_\infty\mathbb{Z} \multimap (!_\infty\mathsf{Set}\langle\mathbb{Z}\rangle \multimap !_\infty\mathsf{List}\langle !_\infty\mathsf{Set}\langle\mathbb{Z}\rangle\rangle))$
$\mathit{hist}'$ = fix $g.\lambda\mathcal{B}.\lambda c.\lambda s.$
          let $y = (c > \mathcal{B})$ in
          case $y$ of $x$ in nil
          else let $(s_1, s_2) = \mathit{setsplit}((\lambda z.z < c), s)$ in
            cons $s_1$ $(g\ \mathcal{B}\ (c + 1)\ s_2)$

$\mathit{hist} :\ !_\infty\mathbb{Z} \multimap (!_1\mathsf{Set}\langle\mathbb{Z}\rangle \multimap !_1\mathsf{List}\langle !_1\mathbb{Z}\rangle)$
$\mathit{hist} = \lambda\mathcal{B}.\lambda s.\ \mathit{map}\ \mathit{size}\ (\mathit{hist}'\ \mathcal{B}\ 0\ s)$
```

Since the histogram function works on sets, we also introduce the following function to transform a list of numbers into a multiset containing all the list elements.

```
$list2set' :\ !_k\mathsf{Set}\langle\mathbb{Z}\rangle \multimap (!_1\mathsf{List}\langle !_{2k}\mathbb{Z}\rangle \multimap !_k\mathsf{Set}\langle\mathbb{Z}\rangle)$
$list2set' = \mathsf{fix}\ g.\lambda s.\lambda l.$
        $\mathsf{unfold}\ l\ \mathsf{as}\ l'\ \mathsf{in}$
          $\mathsf{case}\ l'\ \mathsf{of}\ l''\ \mathsf{in}\ s$
          $\mathsf{else\ let}\ (h,t) = l''\ \mathsf{in}$
             $g\ (s \cup (\mathbb{Z}2set\ h))\ t$
        $\mathsf{else}\ '\mathsf{error}'$

$list2set :\ !_1\mathsf{List}\langle !_{2k}\mathbb{Z}\rangle \multimap !_k\mathsf{Set}\langle\mathbb{Z}\rangle$
$list2set = \lambda l.\ list2set'\ \emptyset\ l$
```

The following function is purely used for type-checking reasons and allows us to transform a list of type $!_1\mathsf{List}\langle !_\infty\tau\rangle$ into a list of type $!_\infty\mathsf{List}\langle !_\infty\tau\rangle$. Intuitively, if we can use each list element arbitrarily often, we can also use the list unboundedly often.

```
$explist :\ !_1\mathsf{List}\langle !_\infty\tau\rangle \multimap !_\infty\mathsf{List}\langle !_\infty\tau\rangle$
$explist = \mathsf{fix}\ g.\lambda l.$
        $\mathsf{unfold}\ l\ \mathsf{as}\ l'\ \mathsf{in}$
          $\mathsf{case}\ l'\ \mathsf{of}\ l''\ \mathsf{in\ nil}$
          $\mathsf{else\ let}\ (h,t) = l''\ \mathsf{in}$
             $\mathsf{let}\ t' = g\ t\ \mathsf{in}$
             $\mathsf{cons}\ h\ t'$
        $\mathsf{else}\ '\mathsf{error}'$
```

The following function checks whether an exponential value is contained in a list and if so, returns true and the complete list.

```
$member :$
    $!_\infty\phi \multimap (!_1\mathsf{List}\langle !_\infty\phi\rangle \multimap (!_\infty\mathsf{Bool} \otimes !_1\mathsf{List}\langle !_\infty\phi\rangle))$
$member =$
    $\mathsf{fix}\ g.\lambda c.\lambda l.$
    $\mathsf{unfold}\ l\ \mathsf{as}\ l'\ \mathsf{in}$
      $\mathsf{case}\ l'\ \mathsf{of}\ l''\ \mathsf{in}\ (\mathsf{false}, \mathsf{nil})$
      $\mathsf{else\ let}\ (h,t) = l''\ \mathsf{in}$
        $\mathsf{case}\ c = h\ \mathsf{of}\ x\ \mathsf{in}$
          $(\mathsf{true}, \mathsf{cons}\ h\ t)$
        $\mathsf{else\ let}\ (r,t') = g\ c\ t\ \mathsf{in}$
          $(r, \mathsf{cons}\ h\ t')$
    $\mathsf{else}\ '\mathsf{error}'$
```

The following function removes all duplicates from a list.

```
$remove\_duplicates$ :
    $!_1\mathsf{List}\langle !_\infty \phi\rangle \multimap !_1\mathsf{List}\langle !_\infty \phi\rangle$
$remove\_duplicates$ =
    fix $g.\lambda l.$
    unfold $l$ as $l'$ in
        case $l'$ of $l''$ in nil
        else
            let $(h, t) = l''$ in
            let $(m, t') = member\ h\ t$ in
            case $m$ of $m'$ in $g\ t'$
            else
                let $t'' = g\ t'$ in
                cons $h\ t''$
    else 'error'
```

The following function takes a list l of values of an option type and returns the list consisting of all x such that some x was in the list l.

```
$remove_{\mathsf{none}}$ :
    $!_1\mathsf{List}\langle !_1\mathsf{Option}\langle\tau\rangle\rangle \multimap !_1\mathsf{List}\langle\tau\rangle$
$remove_{\mathsf{none}}$ =
    fix $g.\lambda l.$
    unfold $l$ as $l'$ in
        case $l'$ of $l''$ in nil
        else
            let $(h, t) = l''$ in
            case $h$ of $h'$ in $g\ t$
            else cons $h'$ $(g\ t)$
    else 'error'
```

Main protocol functions. We now define the main functions of the protocol. Since we assume the publisher to be malicious and thus under the control of the attacker, we will not model it but instead expect the attacker to perform some arbitrary operations of his choosing instead.

To keep the presentation as intuitive as possible, we model the protocol for a single client database query q, which we assume to be a 1-sensitive signature function of type $!_k\mathsf{DB} \multimap !_k\mathbb{Z}$. Furthermore, we we assume the query to be answered by one bucket (out of $\mathcal{B}$ many). Multiple queries and queries that allow multiple buckets as a result can be modeled similarly.

We model the *NTWA* protocol as a function that takes the secret database D as an input and returns a tuple containing the query identifier, the public key of the aggregator, and the functions that model the behavior of the client *client* and the data aggregator *agg*. The code of the protocol implementation is given in Figure 4.2.

Note that in the definition of ϵ-DDP, we consider an opponent O that takes the result of *NTWA D* as an input. In particular, this means that *NTWA D* needs to be given a type $\mathsf{Un} \in \mathcal{OPP}$, so that the result is available to the opponent. Once the opponent is given a query function, he can use it arbitrarily often.

We model the use of a client database in our protocol as described below. If we were to simply use the confidential (affine) client database D in the function that describes the behavior of the client *client*, the environment under which this function were to type-check would not be exponential, meaning we would fail in assigning an opponent type to the client function. Instead, we store D in a affine reference. The reference itself is exponential - which means that it can be used arbitrarily often (and the client function type-checked under an exponential environment) - but its content is not. After retrieving a database once from a reference, the reference will be "empty", so trying to access the secret a second time (if the query is called more than once) will fail.

Note that this corresponds to a real world setting. In general, ϵ-differential privacy only holds for one call to the query. Repeating the query n-times results in a weaker guarantee of $n \cdot \epsilon$-differential privacy, as discussed in Section 4.5.

We now explain how to type-check the overall protocol *NTWA* and its participant functions *client*, *agg* with the following types:

$$\begin{aligned} NTWA &: \ !_2\mathsf{DB} \multimap (\mathsf{Un} \multimap (!_\infty\mathbb{Z} \multimap \mathsf{Un})) \\ client &: \ !_\infty(\mathsf{Un} \multimap \mathsf{Un}) \\ agg &: \ !_\infty(\mathsf{Un} \multimap !_\infty\mathsf{List}\langle !_\infty\mathbb{Z}\rangle) \end{aligned}$$

Note that all types τ that contain only exponential replication indexes (e.g., the types of the *client* and of the *agg* function, as well as the type of the output of *NTWA*) are both super- and subtype of $\mathsf{Un} \in \mathcal{OPP}$.

Remark: The factor 2 in the type of the *NTWA* function and in the types specified below is a minor artifact of our implementation in which we move from lists of integers to sets.

The protocol *NTWA* first sets a flag to activate the query and stores the database in the manner described above. It then creates the enc-/ and decryption keys ek_A $:!_\infty\mathsf{EKey}\langle \mathsf{Q_{Id}} \otimes !_2\mathbb{Z}\rangle$ and k_A $:!_\infty\mathsf{DKey}\langle \mathsf{Q_{Id}} \otimes !_2\mathbb{Z}\rangle$, respectively, which are used to encrypt and decrypt the client's query result. The encryption key will be used in the code of the *client* function, where it is used to encrypt the query identifier (of type $\mathsf{Q_{Id}}$) together with the result (of type $!_2\mathbb{Z}$) of executing query q on the client database D. The encryption produces a public ciphertext of type $!_\infty\mathbb{Z} <: \mathsf{Un}$ that is returned to the opponent.

Furthermore, if the query end time has not yet passed (flag false) the *agg* function removes all duplicates of the ciphertexts it received and then uses its decryption key to decrypt and check the ciphertexts using the guarded decryption function $dec\&check_{NTWA}$. The decryption will succeed at most once per ciphertext, in which case it returns a secret value of type $!_2\mathbb{Z}$. The list of all decrypted query results is transformed into a histogram of type $!_1\mathsf{List}\langle !_1\mathbb{Z}\rangle$. We add noise to each histogram bucket and subtract the offset. Due to the discrete Laplace sanitization mechanism, the resulting list is of type $!_1\mathsf{List}\langle !_\infty\mathbb{Z}\rangle$ and can be transformed into a list that has the public type $\underbrace{!_\infty\mathsf{List}\langle !_\infty\mathbb{Z}\rangle}_{<:\mathsf{Un}}$ which can be published.

Note that the keys have an exponential type. This in particular means that the client and aggregator functions will be type-checked under an exponential typing environment that contains the exponential keys as well as exponential functions to access the affine references. We can apply rule $!_k$I to give each participant function the above specified public types. We can thus return the two server functions together with the public encryption key of the aggregator and the public query identifier to the opponent.

As we have seen, the overall protocol function *NTWA* type-checks with type $!_2\mathsf{DB} \multimap (\mathsf{Un} \multimap (!_\infty\mathbb{Z} \multimap \mathsf{Un})$. Thus, by Theorem 4.1 we know that the protocol provides 2ϵ-distributed differential privacy.

4.9 Extension to other noise mechanisms

The discrete Laplace noise addition and its continuous counterpart are well-established mechanisms for achieving differential privacy but of course not the only ones. In this section we show how our framework can easily be extended to other privacy mechanisms, for which security has been shown independently, e.g., manually or using CertiPriv [38].

Remark (Limitations of finite-precision semantics). We would like to point out that while the theoretical definitions of sanitization mechanisms for DP often operate on idealized infinite-precision semantics, on an implementation level the semantics is necessarily finite-precision. As we have stated before, due to the physical limitations of actual machines we tacitly assume all types and the constants exported by the signature to range over discrete domains. While our system so far considered a mechanism for integers, we note that to include infinite-precision types in the signature they must be expressed in some discrete approximation. For instance, to include reals, they must be rounded according to some level of precision and they must be encoded in some fixed- or floating-point representation of $\mathbb{R}$.

In their works, Mironov [119] and Gazeau et al. [120] show that this mismatch between idealized mechanisms and their finite-precision implementations gives rise to several attacks. The authors show that approximation errors of finite-precision representation of reals can lead to the disclosure of secrets, even if the underlying sanitization mechanisms have been proven to provide differential privacy in an idealized infinite-precision setting. Both papers also provide solutions to fix such privacy breaches for a large class of sanitization mechanisms. The solutions are based on different forms of rounding and truncation and provide a limited (but acceptable) variant of differential privacy.

We would like to emphasize that these results should be taken into account when including a differentially private mechanism that might be affected by these limitations in a finite-precision environment.

```
$NTWA :\ !_2\mathsf{DB} \multimap (\mathsf{Un} \multimap (!_\infty\mathbb{Z} \multimap \mathsf{Un}))$
$NTWA = \lambda D.ek_P.\lambda off.$
        let $define\ library\ \mathcal{L}$ in
        let $q_{id} = 1$ in
        let $(r_{ended}, w_{ended}) = \mathsf{ref}_{!_\infty\mathsf{Bool}}$ in
        let $\_ = w_{ended}$ false in
        let $(r_C, w_C) = \mathsf{ref}_{!_2\mathsf{DB}}$ in
        let $store_C = w_C\ D$ in
        let $k_A = mkDKey\ ()$ in
        let $ek_A = mkEKey\ k_A$ in
        $(q_{id}, ek_A, client, agg)$


$client :\ !_\infty(\mathsf{Un} \multimap \mathsf{Un})$
$client = \lambda ek_P.$
        let $d_C = r_C\ ()$ in
        case $d_C$ of $d'_C$ in 'error'
        else
            let $res = q\ d'_C$ in
            let $res' = aenc\ ek_A\ (q_{id}, res)$ in
            let $res'' = aenc\ ek_P\ (q_{id}, res')$ in
            $res''$


$agg :\ !_\infty(\mathsf{Un} \multimap !_\infty\mathsf{List}\langle !_\infty\mathbb{Z}\rangle)$
$agg = \lambda w.$
        let $ended = r_{ended}\ ()$ in
        case $ended$ of $e$ in
             let $(i, l) = w$ in
             let $id_{ok} = (i = q_{id})$ in
             case $id_{ok}$ of $id'_{ok}$ in
                 let $l_{unique} = remove\_duplicates\ l$ in
                 let $f_{check} = \lambda x.(dec\&check_{NTWA}\ k_A\ x\ (x, q_{id}))$ in
                 let $l_{valid} = remove_{\mathsf{none}}\ (map\ f_{check}\ l_{unique})$ in
                 let $s = list2set\ l_{valid}$ in
                 let $l_{hist} = hist\ \mathcal{B}\ s$ in
                 let $f_N = \lambda x.((\mathsf{add\_noise}^{\epsilon}_{\mathbb{Z}\to\mathbb{Z}}\ x) - off)$ in
                 let $l_{noise} = map\ f_N\ l_{hist}$ in
                 let $finished = r_{ended}$ true in
                 $explist\ l_{noise}$
             else 'error'
        else 'error'
```

Figure 4.2: Code of the *NTWA* protocol

4.9.1 A general extension

We now extend the type system to include a general mechanism primitive that can be instantiated with any privacy mechanism for which security has been shown independently, e.g., manually or using CertiPriv [38]. The revised type system is thus easily extendable and the later inclusion of further noise mechanisms through the general mechanism primitive will not require additional soundness proofs.

We first show the extension of DF7 and then show how to instantiate the general noise primitive with the discrete Laplace mechanism, thus rendering the respective hard-coded primitive $\mathsf{add\_noise}^{s}_{\mathbb{Z}\to\mathbb{Z}}\ M$ obsolete (though convenient). Furthermore, we demonstrate how to exemplarily include two other sanitization mechanisms into the system using the new primitive.

Extending the syntax and semantics. We now show how to extend the syntax and semantics of our calculus to include arbitrary sanitization mechanisms $X^{s}_{b_1\to b_2}$. Here s denotes the privacy parameter of the mechanism that takes inputs of type b_1 and returns sanitized values of type b_2. The following primitive is added to the set of expressions:

$$\mathsf{san\_}X^{s}_{b_1\to b_2}\ M.$$

It is annotated with the parameter s and the appropriate type $b_1 \to b_2$, where both the domain b_1 and the range b_2 of the mechanism are base types in our type system. Its non-deterministic semantics is defined similarly to that of the $\mathsf{add\_noise}^{s}_{\mathbb{Z}\to\mathbb{Z}}$ primitive:

$$[S, \mathsf{san\_}X^{s}_{b_1\to b_2}\ c_1] \xrightarrow{\textsc{San}\ (X^{s}_{b_1\to b_2}, c_1, c_2)}_{p} [S, c_2],$$

where $p = Pr[X^{s}_{b_1\to b_2}(c_1) = c_2]$ and $c_i : b_i \in \Sigma$. Intuitively, $\mathsf{san\_}X^{s}_{b_1\to b_2}\ c_1$ reduces to some constant value c_2 in the query range, where c_2 is drawn according to the distribution $X^{s}_{b_1\to b_2}(c_1)$.

Here, the safety parameter s has the same function as its counterpart in DLap^{s}. Intuitively, we will assume it to be set to $s := \epsilon$. We assume each mechanism $X^{s}_{b_1\to b_2}$ that we include into our system as $\mathsf{san\_}X^{s}_{b_1\to b_2}$ to provide s-differential privacy (for instance, by relying on previous security results by the mechanisms' developers).

Extending the type system. To type-check the general sanitization mechanism primitive, the following rule is added to the type system

$$\text{(San)}\quad \frac{\Gamma \vdash M :!_1 b_1}{\Gamma \vdash \mathsf{san\_}X^{s}_{b_1\to b_2}\ M :!_\infty b_2}.$$

Here, the argument M of the mechanism that is parameterized by its type $b_1 \to b_2$ must be of the domain type b_1 of the query with replication factor $!_1$ (since it might be private). The sanitized result is then given the query range type b_2, replicated by a factor of $!_\infty$, since it may be published.

The corresponding algorithmic typing rule (San Alg) follows the same principle and is listed below:

$$\text{(San Alg)}\quad \frac{\Gamma \vdash_{\mathsf{alg}} M :!_1 b_1; \Gamma'}{\Gamma \vdash_{\mathsf{alg}} \mathsf{san\_}X^{s}_{b_1\to b_2}\ M :!_\infty b_2; \Gamma'}.$$

Adapting the theorem. We need to modify Theorem 4.1 to accommodate the occurrences of other mechanism primitives.The revised theorem is given below and requires $\mathsf{san}\_X^s_{b_1\to b_2}\ M$ to be annotated with an appropriate security parameter s to guarantee $\epsilon/k, \tau$-differential privacy:

Revision 4.1 (of Theorem 4.1)**.** *For all $k \in \mathbb{R}^{>0}$, all types τ,and all closed expressions P such that the following conditions hold:*

- *the parameter of all noise addition primitives occurring in P is set to $s := e^{\epsilon/k}$ (i.e., they are of the form $\mathsf{add\_noise}^{e^{-\epsilon/k}}_{\mathbb{Z}\to\mathbb{Z}}\ M$)*
- *the parameter of all general mechanism primitives $\mathsf{san}\_X^s_{b_1\to b_2}\ M$ for the mechanism $X^s_{b_1\to b_2}$ occurring in P is set to $s := \epsilon/k$ (i.e., they are of the form $\mathsf{san}\_X^{\epsilon/k}_{b_1\to b_2}\ M$) and the respective mechanism $X^s_{b_1\to b_2}$ provides s-differential privacy*
- *$\emptyset \vdash P : \tau \multimap \rho$ for some $\rho \in \mathcal{OPP}$*

P is $\epsilon/k, \tau$-differentially private.

The proof of the revised theorem above does not depend on any specific property of the noise mechanisms $X^s_{b_1\to b_2} \in \mathcal{X}$ that are included as $\mathsf{san}\_X^s_{b_1\to b_2}$, just on the fact that they are assumed to be s-differentially private.

4.9.2 Instantiating the general mechanism primitive.

In the following we will exemplify how to instantiate the general mechanism primitive to capture the discrete Laplace mechanism. Furthermore, we show how to include another sanitization mechanism into the system by using the exponential mechanism and the continuous Laplace mechanism as an example.

Discrete Laplace mechanism. Instead of using the hard-coded noise-addition primitive $\mathsf{add\_noise}^s_{\mathbb{Z}\to\mathbb{Z}}\ M$ we can instantiate the general primitive $\mathsf{san}\_X^r_{b_1\to b_2}\ M$ with the distribution $DL^r_{\mathbb{Z}\to\mathbb{Z}}$ defined as

$$DL^r_{\mathbb{Z}\to\mathbb{Z}}(x) = x + z, \text{ where } z \leftarrow \mathsf{DLap}^{e^{-r}}.$$

As we have seen before, addition of noise drawn according to the discrete Laplace distribution $\mathsf{DLap}^{e^{-r}}$ provides r-differential privacy. We can thus replace $\mathsf{add\_noise}^{e^{-s}}_{\mathbb{Z}\to\mathbb{Z}}\ M$ (which adds noise drawn according to the distribution $\mathsf{DLap}^{e^{-s}}$) with $\mathsf{san}\_DL^s_{\mathbb{Z}\to\mathbb{Z}}\ M$, which has the same privacy guarantees.

Laplace mechanism. One of the oldest and most popular sanitization mechanisms that supports the addition of real-valued noise is the Laplace mechanism [110]. We stress that depending on the choice of finite-precision representations of reals, Mironov [119] and Gazeau et al. [120] have shown that the differential privacy guarantees of the mechanism might not carry over to an implementation. They provide modifications of the mechanism that employ rounding and truncation to guarantee a modified variant of differential privacy. However, for the sake of giving the reader an intuitive introduction of how to include mechanisms into our calculus via the use of the general mechanism primitive, here we show how to include the original Laplace mechanism. The more complex versions by Mironov and Gazeau et al. could be included analogously, depending on the underlying assumed finite-precision representation of reals.

The Laplace mechanism is defined by adding a random number drawn according to the Laplace distribution Lap^s to the correct query result. In an idealized setting, the Laplace mechanism is ϵ-differentially private if the parameter s is set to k/ϵ [110], where k denotes the sensitivity of the query.

An intuitive way to include the Laplace mechanism into the system would thus be to instantiate the general primitive $\mathsf{san\_}X^r_{b_1 \to b_2}\ M$ with the distribution $L^r_{\mathbb{R}\to\mathbb{R}}$ defined as

$$L^r_{\mathbb{R}\to\mathbb{R}}(x) = x + z, \text{ where } z \leftarrow \mathsf{Lap}^{1/r}.$$

Exponential mechanism. There exist many scenarios in which the query result is non-numerical (e.g., queries returning strings or trees) and adding noise leads to nonsensical results or is not well-defined. McSherry and Talwar [112] proposed a general technique to optimize the quality (and exactness) of a query result while still preserving ϵ-differential privacy. Their so-called *exponential mechanism* works on queries on databases D of some type $\mathcal{D}$ that are expected to return a query result a of an arbitrary type $\mathcal{R}$ for which a base measure β exists. Furthermore, it assumes the existence of a *utility function* $q : (\mathcal{D} \times \mathcal{R}) \to \mathbb{R}$, which assigns a real valued score to each possible input-output pair (D, a), thereby measuring the quality of the result a with respect to input D. The higher the score, the better (e.g., more exact) the result. The goal of the mechanism $\varepsilon^\epsilon_q(D)$ is to output the "best" possible result $a \in \mathcal{R}$, while enforcing differential privacy.

The exponential mechanism is defined as follows [112]: For all $q : (\mathcal{D}\times\mathcal{R}) \to \mathbb{R}$ and all base measures β over $\mathcal{R}$ the randomized exponential mechanism $\varepsilon^\epsilon_q(D)$ for $D \in \mathcal{D}$ is defined as

$$\begin{aligned}\varepsilon^\epsilon_q(D) := \ & \text{return } a \in \mathcal{R} \text{ with probability} \\ & \text{proportional to } e^{\epsilon q(D,a)} \cdot \beta(a).\end{aligned}$$

McSherry and Talwar show that this definition captures the entire class of differential privacy mechanisms and give an encoding of Laplace noise addition by choosing an appropriate utility function q. The link between the exponential mechanism and differential privacy is defined as follows: $\varepsilon^\epsilon_q(D)$ gives $2\epsilon\Delta q$-differential privacy. Here, Δq is defined as the largest possible difference in the utility function when applied to two inputs that differ only on a single user's value.

Before showing how one could include the mechanism into our system we again stress that depending on the choice of q, the choice of the types $\mathcal{D}, \mathcal{R}$ and the choice of finite-precision representations, the privacy guarantees of the idealized mechanism might not straightforwardly carry over to our system but might instead require a modification similar to the techniques proposed by Gazeau et al. [120].

To include the mechanism into our system we could instantiate the general primitive $\mathsf{san}\_X^r_{b_1 \to b_2}\ M$ with the respective distribution $(E_q)^r_{b_1 \to b_2}$ defined as

$$(E_q)^r_{b_1 \to b_2} = \varepsilon_q^{r/(2\Delta q)}, \text{ where } q : (b_1 \times b_2) \to \mathbb{R}.$$

By the differential privacy guarantees of the mechanism we know that $\varepsilon_q^{r/(2\Delta q)}$ gives $2(r/(2\Delta q))\Delta q$-, i.e., r-differential privacy.

4.10 Related work

The formal verification of differential privacy has recently received increasing attention by the academic community. Barthe et al. have presented CertiPriv [38], a machine-checked framework for reasoning about differential privacy built on top of the Coq proof assistant. This framework nicely complements our approach, allowing one to derive formal guarantees of differential privacy for a variety of sanitization mechanisms, such as the one based on Laplace noise, whose correctness is instead assumed in our approach. In later work, Barthe et al. [39] have proposed an alternative approach to verify sanitization mechanisms with respect to the weaker notion of ϵ, δ-approximate differential privacy using Hoare logic specifications. Both approaches, however, have not been used to reason about complex cryptographic protocols and network-level attacks and proofs are not automated. While the former restrictions remain, the latter restriction is no longer present in a recent follow-up work [100], in which the authors present a relational refinement type system for the more general verification of mechanism design and approximative differential privacy. Tschantz et al. [121] showed how to verify differential privacy properties based on I/O-automata. They focus on the usage of differentially private sanitization mechanisms within interactive systems, but they do not explicitly consider cryptographic protocols. Recently, Chaudhuri et al. [122] have introduced a generic robustness property for programs that encompasses sensitivity and can be statically enforced even for programs featuring complex branching structures and loops. This could be useful to further enhance the expressivity of our framework.

While this chapter focuses on the notion of differential privacy, we would like to mention some recent works that discuss some limitations of differential privacy [123, 124] and propose alternative notions of privacy for queries on statistical databases, such as relaxations of differential privacy [125], noiseless definitions [126], and zero-knowledge based definitions for social networks [127].

Finally, since the seminal work by Abadi on "secrecy by typing in security protocols" [128], type systems acquired growing popularity in the analysis of cryptographic protocols and their implementations, and they have been applied to statically enforce secrecy definitions based on reachability properties [29, 113], strong secrecy [34] properties based on observational equivalence relations, as well as privacy [129, 130], authenticity [23, 25–27, 50, 69] and authorization policies [31–33, 57, 68] None of these type systems, however, enforces quantitative secrecy properties.

4.11 Conclusion

This chapter introduced the first mechanized verification technique for distributed differential privacy. Our framework comprises a symbolic definition of differential privacy for distributed databases, which takes into account Dolev-Yao intruders, and DF7, an affine, distance-aware type system to verify this property in cryptographic protocol implementations. DF7$_{\text{alg}}$, a sound and complete algorithmic variant of the type system allows for mechanizing our analysis technique. To the best of our knowledge, this is first automated verification technique for cryptographic protocols that supports a quantitative secrecy property. We have evaluated our system on a protocol for non-tracking web analytics and discovered a new attack. We proposed and verified a revised version of the protocol.

We demonstrated the flexibility of our approach by showing how to easily incorporate other sanitization techniques into our framework.

Part IV

Conclusion

5
Conclusion

In this thesis we presented three frameworks for the verification of security protocols and their implementations based on powerful types for security and privacy. In all three cases, our approaches improve the state-of-the-art and allow for the verification of cryptographic protocols and properties that were out of the scope of previous systems.

We first proposed AF7, the first type system that statically enforces the safety of cryptographic protocol implementations with respect to authorization policies expressed in *affine logic*. Affine logic can be used to express resource-aware properties that were out of the scope of previous type-based analysis techniques. AF7 leverages general-purpose theorem proving techniques, and extends previous systems to support affine logic. To protect affine formulas from duplication we introduced the novel notion of "exponential serialization", which AF7 relies on. We demonstrated the effectiveness of AF7 on two case studies, the EPMO and the Kerberos protocols. Furthermore, we proposed a sound and complete algorithmic variant of the system called AF7$_{\text{alg}}$, which is the key to achieving an efficient implementation of our analysis technique.

Second, we presented a novel approach for the automated analysis of e-voting protocols based on refinement type systems. Specifically, we designed a generically applicable logical theory which, based on pre- and post-conditions for security-critical code, guides existing type-checkers towards the verification of two fundamental properties of e-voting protocols, namely, vote privacy and verifiability, which were out of the scope of previous automated analysis techniques (type-based or other). We provided a code-based cryptographic abstraction of the cryptographic primitives commonly used in e-voting protocols, showing how to make the underlying algebraic properties accessible to automated verification through logical refinements. We demonstrated the effectiveness of our approach by developing the first automated analysis of both the mix-net and the homomorphic version of Helios, a popular web-based e-voting protocol, using an off-the-shelf type-checker.

Finally, we proposed a symbolic definition of differential privacy for distributed databases, which takes into account Dolev-Yao intruders and can be used to reason about compromised parties. We then introduced DF7, an affine, distance-aware type system to statically and automatically enforce this notion of distributed differential privacy in cryptographic protocol implementations. DF7 builds on and significantly extends a previous type system [36] for the non-distributed case of differential privacy, which did not need to consider network attackers or compromised parties. We also provided a sound and complete algorithmic variant of our type system called $\mathrm{DF7}_{\mathrm{alg}}$ and tested our analysis technique on a recently proposed protocol for privacy-preserving web analytics: we discovered a new attack acknowledged by the authors, proposed a fix, and successfully type-checked the revised variant.

6 Directions for Future Research

Based on the work presented in this thesis there are several directions for future research that appear to be of great interest.

One crucial building block that is missing to reliably automate the type-checking approach introduced in Chapter 2 and that seems to be of independent interest to both the security and the formal methods community is the development of a powerful and expressive automated theorem prover for affine logic.

Once such a tool exists, it would be interesting for the development of a type-checker that implements AF7 to investigate how to reduce the need for manual type annotations, e.g., by taking advantage of recent research on type inference in intuitionistic linear logic [131]. A similar approach could prove helpful in the implementation of a type-checker for DF7.

A natural extension of our framework for e-voting is to extend the logical theory to handle even more cryptographic primitives and security properties. For instance, a next step could be to investigate *strong* end-to-end verifiability, which additionally takes the notion of eligibility verifiability into account. This stronger notion is not satisfied by Helios, but the Helios-C protocol [95] was designed to achieve this property, providing an interesting case study for our approach.

Furthermore, it would be interesting to apply our approach to more e-voting protocols, e.g., the protocol recently deployed in Norway for a political election. The privacy of this protocol was analyzed in [90], but due to the algebraic properties of the encryption, the proof was completely done by hand. Our approach looks promising to enable automation of proofs for this and other protocols.

A further exciting direction for future research would be to investigate how to extend the DF7 type system to deal with sanitization mechanisms that provide the weaker notion of ϵ, δ-approximate differential privacy, for instance using some insights from the recent work by Barthe et al. [39] that considers such mechanisms.

Bibliography

[1] M. Bugliesi, S. Calzavara, F. Eigner, and M. Maffei, "Resource-aware Authorization Policies in Statically Typed Cryptographic Protocols," in *Proc. 24th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, 2011, pp. 83–98.

[2] ——, "Affine Refinement Types for Authentication and Authorization," in *Proc. 7th International Symposium on Trustworthy Global Computing (TGC)*, ser. Lecture Notes in Computer Science, vol. 8191. Springer-Verlag, 2012, pp. 19–33.

[3] ——, "Logical Foundations of Secure Resource Management in Protocol Implementations," in *Proc. 2nd International Conference on Principles of Security and Trust (POST)*, ser. Lecture Notes in Computer Science, vol. 7796. Springer-Verlag, 2013, pp. 105–125.

[4] ——, "Affine Refinement Types for Secure Distributed Programming," *ACM Transactions on Programming Languages and Systems*, vol. 37, no. 4, pp. 11:1–11:66, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2743018

[5] V. Cortier, F. Eigner, S. Kremer, M. Maffei, and C. Wiedling, "Type-Based Verification of Electronic Voting Protocols," in *Proc. 4th International Conference on Principles of Security and Trust (POST)*, ser. Lecture Notes in Computer Science, vol. 9036. Springer-Verlag, 2015, pp. 303–323.

[6] ——, "Type-Based Verification of Electronic Voting Protocols," Cryptology ePrint Archive, Report 2015/039, 2015.

[7] F. Eigner and M. Maffei, "Differential Privacy by Typing in Security Protocols," in *Proc. 26th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, 2013, pp. 272–286.

[8] F. Eigner, A. Kate, M. Maffei, F. Pampaloni, and I. Pryvalov, "Differentially Private Data Aggregation with Optimal Utility," in *Proc. 30th Annual Computer Security Applications Conference (ACSAC)*. ACM Press, 2014, pp. 316–325.

[9] ——, "Differentially Private Data Aggregation with Optimal Utility," Cryptology ePrint Archive, Report 2014/482, 2014.

[10] ——, “Achieving Optimal Utility for Distributed Differential Privacy Using SMPC,” in *Applications of Secure Multiparty Computation*, ser. Cryptology and Information Security Series. IOS Press, 2015, vol. 13, ch. 5, pp. 81 – 105.

[11] I. E. Akkus, R. Chen, M. Hardt, P. Francis, and J. Gehrke, “Non-tracking Web Analytics,” in *Proc. 19th ACM Conference on Computer and Communications Security (CCS).* ACM Press, 2012, pp. 687–698.

[12] G. Lowe, “Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR,” in *Proc. 2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, vol. 1055. Springer-Verlag, 1996, pp. 147–166.

[13] D. Wagner and B. Schneier, “Analysis of the SSL 3.0 Protocol,” in *Proc. 2nd USENIX Workshop on Electronic Commerce.* USENIX Association, 1996, pp. 29–40.

[14] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel, “Attacking and Fixing PKCS#11 Security Tokens,” in *Proc. 17th ACM Conference on Computer and Communications Security (CCS).* ACM Press, 2010, pp. 260–269.

[15] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra, “Formal Analysis of SAML 2.0 Web Browser Single Sign-on: Breaking the SAML-based Single Sign-on for Google Apps,” in *Proc. 6th ACM Workshop on Formal Methods in Security Engineering (FMSE).* ACM Press, 2008, pp. 1–10.

[16] B. Blanchet, “An Efficient Cryptographic Protocol Verifier Based on Prolog Rules.” in *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW).* IEEE Computer Society Press, 2001, pp. 82–96.

[17] M. Backes, S. Lorenz, M. Maffei, and K. Pecina, “The CASPA Tool: Causality-Based Abstraction for Security Protocol Analysis,” in *Proc. Computer Aided Verification’08 (CAV)*, ser. Lecture Notes in Computer Science, vol. 5123. Springer-Verlag, 2008, pp. 419–422.

[18] C. Cremers, “The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols,” in *Proc. Computer Aided Verification’08 (CAV)*, ser. Lecture Notes in Computer Science, vol. 5123, 2008, pp. 414–418.

[19] M. Backes, A. Cortesi, and M. Maffei, “Causality-based Abstraction of Multiplicity in Cryptographic Protocols,” in *Proc. 20th IEEE Symposium on Computer Security Foundations (CSF).* IEEE Computer Society Press, 2007, pp. 355–369.

[20] B. Blanchet, M. Abadi, and C. Fournet, “Automated Verification of Selected Equivalences for Security Protocols,” *Journal of Logic and Algebraic Programming*, vol. 75, no. 1, pp. 3–51, 2008.

[21] R. Chadha, Ş. Ciobâcă, and S. Kremer, “Automated Verification of Equivalence Properties of Cryptographic Protocols,” in *Proc. 21st European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 7211. Springer-Verlag, 2012, pp. 108–127.

[22] V. Cheval, “APTE: An Algorithm for Proving Trace Equivalence,” in *Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, vol. 8413. Springer-Verlag, 2014, pp. 587–592.

[23] A. D. Gordon and A. Jeffrey, “Types and Effects for Asymmetric Cryptographic Protocols,” *Journal of Computer Security*, vol. 12, no. 3, pp. 435–484, 2004.

[24] M. Bugliesi, R. Focardi, and M. Maffei, “Principles for Entity Authentication,” in *Proc. 5th International Conference Perspectives of System Informatics (PSI)*, ser. Lecture Notes in Computer Science, vol. 2890. Springer-Verlag, 2003, pp. 294–306.

[25] ——, “Authenticity by Tagging and Typing,” in *Proc. 2nd ACM Workshop on Formal Methods in Security Engineering (FMSE).* ACM Press, 2004, pp. 1–12.

[26] ——, “Analysis of Typed Analyses of Authentication Protocols,” in *Proc. 18th IEEE Computer Security Foundations Workshop (CSFW).* IEEE Computer Society Press, 2005, pp. 112–125.

[27] ——, “Dynamic Types for Authentication,” *Journal of Computer Security*, vol. 15, no. 6, pp. 563–617, 2007.

[28] M. Backes, A. Cortesi, R. Focardi, and M. Maffei, “A Calculus of Challenges and Responses,” in *Proc. 5th ACM Workshop on Formal Methods in Security Engineering (FMSE).* ACM Press, 2007, pp. 101–116.

[29] R. Focardi and M. Maffei, “Types for Security Protocols,” in *Formal Models and Techniques for Analyzing Security Protocols*, ser. Cryptology and Information Security Series. IOS Press, 2011, vol. 5, ch. 7, pp. 143–181.

[30] M. Backes, M. P. Grochulla, C. Hriţcu, and M. Maffei, “Achieving security despite compromise using zero-knowledge,” in *Proc. 22nd IEEE Symposium on Computer Security Foundations (CSF).* IEEE Computer Society Press, 2009, pp. 308–323.

[31] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, “Refinement Types for Secure Implementations,” *ACM Transactions on Programming Languages and Systems*, vol. 33, no. 2, pp. 8:1–8:45, 2011.

[32] M. Backes, C. Hriţcu, and M. Maffei, "Union and Intersection Types for Secure Protocol Implementations," in *Proc. Theory of Security and Applications (TOSCA)*, ser. Lecture Notes in Computer Science, vol. 6993. Springer-Verlag, 2011, pp. 1–28.

[33] ——, "Union and Intersection Types for Secure Protocol Implementations," *Journal of Computer Security*, vol. 22, no. 2, pp. 301–353, 2014.

[34] C. Fournet, M. Kohlweiss, and P.-Y. Strub, "Modular Code-Based Cryptographic Verification," in *Proc. 18th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, 2011, pp. 341–350.

[35] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang, "Secure Distributed Programming with Value-Dependent Types," in *Proc. 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM Press, 2011, pp. 266–278.

[36] J. Reed and B. C. Pierce, "Distance Makes the Types Grow stronger: A Calculus for Differential Privacy," in *Proc. 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM Press, 2010, pp. 157–168.

[37] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce, "Linear Dependent Types for Differential Privacy," in *Proc. 40th Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2013, pp. 357–370.

[38] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin, "Probabilistic Relational Reasoning for Differential Privacy," in *Proc. 39th Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2012, pp. 97–110.

[39] G. Barthe, M. Gaboardi, E. J. G. Arias, J. Hsu, C. Kunz, and P. Strub, "Proving Differential Privacy in Hoare Logic," in *Proc. 27th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, 2014, pp. 411–424.

[40] S. Calzavara, "Static Verification and Enforcement of Authorization Policies," Ph.D. dissertation, Università Ca' Foscari Venezia, Italy, 2013.

[41] P. C. Chapin, C. Skalka, and X. S. Wang, "Authorization in Trust Management: Features and Foundations," *ACM Computing Surveys*, vol. 40, no. 3, pp. 9:1–9:48, 2008.

[42] J.-Y. Girard, "Linear Logic: Its Syntax and Semantics," in *Advances in Linear Logic*, ser. London Mathematical Society Lecture Note Series, vol. 22, 1995, pp. 3–42.

[43] A. S. Troelstra, "Lectures on Linear Logic," CSLI Stanford, Lecture Notes Series Nr. 29, 1992.

[44] J. D. Guttman, F. J. Thayer, J. A. Carlson, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen, "Trust Management in Strand Spaces: A Rely-Guarantee Method," in *Proc. 13th European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 2986. Springer-Verlag, 2004, pp. 325–339.

[45] J. G. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," in *Proc. USENIX Summer Conference.* USENIX Association, 1988, pp. 191–202.

[46] C. Fournet, A. D. Gordon, and S. Maffeis, "A Type Discipline for Authorization Policies," in *Proc. 14th European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 3444. Springer-Verlag, 2005, pp. 141–156.

[47] ——, "A Type Discipline for Authorization in Distributed Systems," in *Proc. 20th IEEE Symposium on Computer Security Foundations (CSF).* IEEE Computer Society Press, 2007, pp. 31–45.

[48] N. 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)*, vol. 75, no. 5, pp. 381 – 392, 1972.

[49] A. Tiu and A. Momigliano, "Cut Elimination for a Logic with Induction and Co-induction," *Journal of Applied Logic*, vol. 10, no. 4, pp. 330–367, 2012.

[50] A. D. Gordon and A. Jeffrey, "Authenticity by Typing for Security Protocols," *Journal of Computer Security*, vol. 11, no. 4, pp. 451–519, 2003.

[51] Y. Mandelbaum, D. Walker, and R. Harper, "An Effective Theory of Type Refinements," in *Proc. 8th ACM SIGPLAN International Conference on Functional Programming (ICFP).* ACM Press, 2003, pp. 213–225.

[52] M. Fähndrich and R. DeLine, "Adoption and focus: practical linear types for imperative programming," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM Press, 2002, pp. 13–24.

[53] J. Morris, "Protection in Programming Languages," *Communications of the ACM*, vol. 16, no. 1, pp. 15–21, 1973.

[54] E. Sumii and B. Pierce, "A Bisimulation for Dynamic Sealing," *Theoretical Computer Science*, vol. 375, no. 1-3, pp. 169–192, 2007.

[55] M. Backes, M. Maffei, and D. Unruh, “Computationally Sound Verification of Source Code,” in *Proc. 17th ACM Conference on Computer and Communications Security (CCS).* ACM Press, 2010, pp. 387–398.

[56] N. Tomura, “llprover - A Linear Logic Prover,” 1995, http://bach.istc.kobe-u.ac.jp/llprover/.

[57] K. Bhargavan, C. Fournet, and A. D. Gordon, “Modular Verification of Security Protocol Code by Typing,” in *Proc. 37th Symposium on Principles of Programming Languages (POPL).* ACM Press, 2010, pp. 445–456.

[58] K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. J. Leifer, “Cryptographic Protocol Synthesis and Verification for Multiparty Sessions,” in *Proc. 22nd IEEE Symposium on Computer Security Foundations (CSF).* IEEE Computer Society Press, 2009, pp. 124–140.

[59] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, “Verified Interoperable Implementations of Security Protocols,” *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 1, pp. 1–61, 2008.

[60] R. Milner, “Functions as Processes,” *Mathematical Structures in Computer Science*, vol. 2, no. 2, pp. 119–141, 1992.

[61] K. Bierhoff and J. Aldrich, “Modular Typestate Checking of Aliased Objects,” in *Proc. 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA).* ACM Press, 2007, pp. 301–320.

[62] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and E. Tanter, “First-Class State Change in Plaid,” in *Proc. 26th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA).* ACM Press, 2011, pp. 713–732.

[63] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff, “A Type System for Borrowing Permissions,” in *Proc. 39th Symposium on Principles of Programming Languages (POPL).* ACM Press, 2012, pp. 557–570.

[64] J. Tov and R. Pucella, “Stateful Contracts for Affine Types,” in *Proc. 19th European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2010, vol. 6012, pp. 550–569.

[65] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron, “The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications,” in *Proc. Computer Aided Verification’05 (CAV)*, ser. Lecture Notes in Computer Science, vol. 3576. Springer-Verlag, 2005, pp. 281–285.

[66] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin, "The TAMARIN Prover for the Symbolic Analysis of Security Protocols," in *Proc. Computer Aided Verification'13 (CAV)*, ser. Lecture Notes in Computer Science, vol. 8044. Springer-Verlag, 2013, pp. 696–701.

[67] B. Blanchet, "Using Horn Clauses for Analyzing Security Protocols," in *Formal Models and Techniques for Analyzing Security Protocols*, ser. Cryptology and Information Security. IOS Press, 2011, vol. 5, ch. 7, pp. 86–111.

[68] M. Backes, C. Hriţcu, and M. Maffei, "Type-checking Zero-knowledge," in *Proc. 15th ACM Conference on Computer and Communications Security (CCS).* ACM Press, 2008, pp. 357–370.

[69] M. Bugliesi, R. Focardi, and M. Maffei, "Compositional Analysis of Authentication Protocols," in *Proc. 13th European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 2986. Springer-Verlag, 2004, pp. 140–154.

[70] M. Maffei, "Tags for Multi-Protocol Authentication," in *Proc. 2nd International Workshop on Security Issues in Coordination Models, Languages, and Systems (SECCO)*, ser. Electronic Notes on Theoretical Computer Science. Elsevier Science Publishers Ltd., 2004, pp. 55–63.

[71] M. Abadi and C. Fournet, "Mobile Values, New Names, and Secure Communication," in *Proc. 28th Symposium on Principles of Programming Languages (POPL).* ACM Press, 2001, pp. 104–115.

[72] C. Wiedling, "Formal Verification of Advanced Families of Security Protocols," Ph.D. dissertation, LORIA, Nancy, France, 2014.

[73] S. Estehghari and Y. Desmedt, "Exploiting the Client Vulnerabilities in Internet E-voting Systems: Hacking Helios 2.0 as an Example," in *Proc. Electronic Voting Technology Workshop/ Workshop on Trustworthy Elections (EVT/WOTE).* USENIX Association, 2010.

[74] V. Cortier and B. Smyth, "Attacking and Fixing Helios: An Analysis of Ballot Secrecy," in *Proc. 24th IEEE Symposium on Computer Security Foundations (CSF).* IEEE Computer Society Press, 2011, pp. 297–311.

[75] R. Küsters, T. Truderung, and A. Vogt, "Clash Attacks on the Verifiability of E-Voting Systems," in *Proc. 33rd IEEE Symposium on Security & Privacy (S&P).* IEEE Computer Society Press, 2012, pp. 395–409.

[76] S. Kremer and M. D. Ryan, "Analysis of an Electronic Voting Protocol in the Applied Pi Calculus," in *Proc. 14th European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 3444. Springer-Verlag, 2005, pp. 186–200.

[77] M. Backes, C. Hriţcu, and M. Maffei, "Automated Verification of Remote Electronic Voting Protocols in the Applied Pi-calculus," in *CSF'08.* IEEE Computer Society Press, 2008, pp. 195–209.

[78] B. Smyth, M. D. Ryan, S. Kremer, and M. Kourjieh, "Towards automatic analysis of election verifiability properties," in *Proc. Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS)*, ser. Lecture Notes in Computer Science, vol. 6186, 2010, pp. 165–182.

[79] M. Arapinis, V. Cortier, S. Kremer, and M. D. Ryan, "Practical Everlasting Privacy," in *Proc. 2nd International Conference on Principles of Security and Trust (POST)*, ser. Lecture Notes in Computer Science, vol. 7796. Springer-Verlag, 2013, pp. 21–40.

[80] J. D. Cohen and M. J. Fischer, "A Robust and Verifiable Cryptographically Secure Election Scheme," in *Proc. 26th IEEE Symposium on Foundations of Computer Science (FOCS).* IEEE Computer Society Press, 1985, pp. 372–382.

[81] R. Cramer, R. Gennaro, and B. Schoenmakers, "A Secure and Optimally Efficient Multi-Authority Election Scheme," in *Advances in Cryptology - Proc. 16th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, ser. Lecture Notes in Computer Science, vol. 1233. Springer-Verlag, 1997, pp. 103–118.

[82] B. Adida, "Helios: Web-based Open-audit Voting," in *Proc. USENIX Conference.* USENIX Association, 2008, pp. 335–348.

[83] S. Delaune, S. Kremer, and M. D. Ryan, "Verifying Privacy-Type Properties of Electronic Voting Protocols," *Journal of Computer Security*, vol. 17, no. 4, pp. 435–487, 2009.

[84] IACR. Elections page at http://www. iacr.org/elections/.

[85] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, vol. 4963. Springer-Verlag, 2008, pp. 337–340.

[86] S. Kremer, M. D. Ryan, and B. Smyth, "Election Verifiability in Electronic Voting Protocols," in *Proc. 15th European Symposium on Research in Computer Security (ESORICS)*, ser. Lecture Notes in Computer Science, vol. 6345. Springer-Verlag, 2010, pp. 389–404.

[87] R. Küsters, T. Truderung, and A. Vogt, "Accountabiliy: Definition and Relationship to Verifiability," in *CCS'10.* ACM Press, 2010, pp. 526–535.

[88] V. Cortier, F. Eigner, S. Kremer, M. Maffei, and C. Wiedling, “Source-code of Helios Analysis,” http://sps.cs.uni-saarland.de/voting.

[89] G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Z. Béguelin, “Probabilistic Relational Verification for Cryptographic Implementations,” in *Proc. 41st Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2014, pp. 193–206.

[90] V. Cortier and C. Wiedling, “A formal analysis of the Norwegian E-voting protocol,” in *Proc. 1st International Conference on Principles of Security and Trust (POST)*, ser. Lecture Notes in Computer Science, vol. 7215. Springer-Verlag, Mar. 2012, pp. 109–128.

[91] A. Fujioka, T. Okamoto, and K. Ohta, “A Practical Secret Voting Scheme for Large Scale Elections,” in *Advances in Cryptology - Proc. Workshop on the Theory and Application of Cryptographic Techniques (AUSCRYPT)*, ser. Lecture Notes in Computer Science, vol. 718. Springer-Verlag, 1992, pp. 244–251.

[92] R. Küsters, T. Truderung, and A. Vogt, “A Game-Based Definition of Coercion-Resistance and its Applications,” in *CSF’10*. IEEE Computer Society Press, 2010, pp. 122–136.

[93] D. Bernhard, V. Cortier, O. Pereira, B. Smyth, and B. Warinschi, “Adapting Helios for Provable Ballot Secrecy,” in *Proc. 16th European Symposium on Research in Computer Security (ESORICS)*, ser. Lecture Notes in Computer Science, vol. 6879. Springer-Verlag, 2011, pp. 335–354.

[94] A. Juels, D. Catalano, and M. Jakobsson, “Coercion-Resistant Electronic Elections,” in *Towards Trustworthy Elections: New Directions in Electronic Voting*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2010, vol. 6000, pp. 37–63.

[95] V. Cortier, D. Galindo, S. Glondu, and M. Izabachène, “Election Verifiability for Helios under Weaker Trust Assumptions,” in *Proc. 19th European Symposium on Research in Computer Security (ESORICS)*, ser. Lecture Notes in Computer Science, vol. 8713. Springer-Verlag, 2014, pp. 327–344.

[96] R. Küsters, T. Truderung, and A. Vogt, “Verifiability, Privacy, and Coercion-Resistance: New Insights from a Case Study,” in *S&P’11*. IEEE Computer Society, 2011, pp. 538–553.

[97] R. L. Rivest and W. D. Smith, “Three Voting Protocols: ThreeBallot, VAV, and Twin,” in *Proc. USENIX Workshop on Accurate Electronic Voting Technology (EVT)*. USENIX Association, 2007, pp. 16–16.

[98] A. Narayanan and V. Shmatikov, “Robust De-anonymization of Large Sparse Datasets,” in *Proc. 29th IEEE Symposium on Security & Privacy (S&P)*. IEEE Computer Society Press, 2008, pp. 111–125.

[99] C. Dwork, "Differential Privacy," in *Proc. 33rd International Colloquium on Automata, Languages and Programming (ICALP)*, ser. Lecture Notes in Computer Science, vol. 4052. Springer-Verlag, 2006, pp. 1–12.

[100] G. Barthe, M. Gaboardi, E. J. G. Arias, J. Hsu, A. Roth, and P. Strub, "Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy," in *Proc. 42nd Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2015, pp. 55–68.

[101] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor, "Our Data, Ourselves: Privacy Via Distributed Noise Generation," in *Advances in Cryptology - Proc. 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, ser. Lecture Notes in Computer Science, vol. 4004. Springer-Verlag, 2006, pp. 486–503.

[102] A. Beimel, K. Nissim, and E. Omri, "Distributed Private Data Analysis: Simultaneously Solving How and What," in *Advances in Cryptology - Proc. 28th Annual International Cryptology Conference (CRYPTO)*, ser. Lecture Notes in Computer Science, vol. 5157. Springer-Verlag, 2008, pp. 451–468.

[103] I. Mironov, O. Pandey, O. Reingold, and S. P. Vadhan, "Computational Differential Privacy," in *Advances in Cryptology - Proc. 29th Annual International Cryptology Conference (CRYPTO)*, ser. Lecture Notes in Computer Science, vol. 5677. Springer-Verlag, 2009, pp. 126–142.

[104] E. Shi, T.-H. H. Chan, E. G. Rieffel, R. Chow, and D. Song, "Privacy-Preserving Aggregation of Time-Series Data," in *Proc. 18th Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2011.

[105] V. Rastogi and S. Nath, "Differentially Private Aggregation of Distributed Time-Series with Transformation and Encryption," in *Proc. 36th ACM SIGMOD International Conference on Management of Data (SIGMOD-/PODS)*. ACM Press, 2010, pp. 735–746.

[106] J. Hsu, S. Khanna, and A. Roth, "Distributed Private Heavy Hitters," *CoRR - Computing Research Repository*, vol. abs/1202.4910, 2012.

[107] A. McGregor, I. Mironov, T. Pitassi, O. Reingold, K. Talwar, and S. P. Vadhan, "The Limits of Two-Party Differential Privacy," in *Proc. 51th IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society Press, 2010, pp. 81–90.

[108] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke, "Towards Statistical Queries over Distributed Private User Data," in *Proc. 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2012.

[109] S. P. Kasiviswanathan, H. K. Lee, K. Nissim, S. Raskhodnikova, and A. Smith, "What Can We Learn Privately?" in *Proc. 49th IEEE Symposium on Foundations of Computer Science (FOCS).* IEEE Computer Society Press, 2008.

[110] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating Noise to Sensitivity in Private Data Analysis," in *Proc. 3rd Theory of Cryptography Conference (TCC)*, ser. Lecture Notes in Computer Science, vol. 3876. Springer-Verlag, 2006, pp. 265–284.

[111] A. Ghosh, T. Roughgarden, and M. Sundararajan, "Universally Utility-Maximizing Privacy Mechanisms," in *Proc. 41st Annual ACM Symposium on Theory of Computing (STOC).* ACM Press, 2009, pp. 351–360.

[112] F. McSherry and K. Talwar, "Mechanism Design via Differential Privacy," in *Proc. 48th IEEE Symposium on Foundations of Computer Science (FOCS).* IEEE Computer Society Press, 2007, pp. 94–103.

[113] M. Abadi and B. Blanchet, "Secrecy Types for Asymmetric Communication," in *Proc. 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, ser. Lecture Notes in Computer Science, vol. 2030. Springer-Verlag, 2001, pp. 25–41.

[114] B. Blanchet, "Automatic Proof of Strong Secrecy for Security Protocols," in *Proc. 25th IEEE Symposium on Security & Privacy (S&P).* IEEE Computer Society Press, 2004, pp. 86–100.

[115] P. S. Barth, R. S. Nikhil, and Arvind, "M-structures: Extending a parallel, non-strict, functional language with state," in *Proc. 5th Conference on Functional Programming Languages and Computer Architecture, (FPCA)*, ser. Lecture Notes in Computer Science, vol. 523. Springer-Verlag, 1991, pp. 538–568.

[116] T.-H. H. Chan, E. Shi, and D. Song, "Privacy-Preserving Stream Aggregation with Fault Tolerance," in *Proc. 16th Financial Cryptography Conference (FC)*, ser. Lecture Notes in Computer Science, vol. 7397. Springer-Verlag, 2012, pp. 200–214.

[117] F. McSherry, "Privacy Integrated Queries: an Extensible Platform for Privacy-Preserving Data Analysis," in *Proc. 35th ACM SIGMOD International Conference on Management of Data (SIGMOD/PODS).* ACM Press, 2009, pp. 19–30.

[118] I. Cervesato, J. S. Hodas, and F. Pfenning, "Efficient Resource Management for Linear Logic Proof Search," *Theoretical Computer Science*, vol. 232, no. 1-2, pp. 133–163, 2000.

[119] I. Mironov, "On Significance of the Least Significant Bits for Differential Privacy," in *Proc. 19th ACM Conference on Computer and Communications Security (CCS).* ACM Press, 2012, pp. 650–661.

[120] I. Gazeau, D. Miller, and C. Palamidessi, "Preserving differential privacy under finite-precision semantics," in *Proc. 11th International Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL)*, 2013, pp. 1–18.

[121] M. C. Tschantz, D. Kaynar, and A. Datta, "Formal Verification of Differential Privacy for Interactive Systems (Extended Abstract)," *Electronic Notes on Theoretical Computer Science*, vol. 276, pp. 61–79, 2011.

[122] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, "Proving Programs Robust," in *Proc. 19th ACM SIGSOFT Symposium and 13th European Conference on Foundations of Software Engineering (ESEC/FSE).* ACM Press, 2011, pp. 102–112.

[123] D. Kifer and A. Machanavajjhala, "No Free Lunch in Data Privacy," in *Proc. 37th ACM SIGMOD International Conference on Management of Data (SIGMOD/PODS).* ACM Press, 2011, pp. 193–204.

[124] A. Haeberlen, B. C. Pierce, and A. Narayan, "Differential Privacy under Fire," in *Proc. USENIX Conference.* USENIX Association, 2011.

[125] S. P. Kasiviswanathan and A. Smith, "A Note on Differential Privacy: Defining Resistance to Arbitrary Side Information," 2008, cryptology ePrint Archive, Report 2008/144.

[126] R. Bhaskar, A. Bhowmick, V. Goyal, S. Laxman, and A. Thakurta, "Noiseless Database Privacy," in *Advances in Cryptology - Proc. 17th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, ser. Lecture Notes in Computer Science, vol. 7073. Springer-Verlag, 2011, pp. 215–232.

[127] J. Gehrke, E. Lui, and R. Pass, "Towards Privacy for Social Networks: A Zero-Knowledge Based Definition of Privacy," in *Proc. 8th Theory of Cryptography Conference (TCC)*, ser. Lecture Notes in Computer Science, vol. 6597. Springer-Verlag, 2011, pp. 432–449.

[128] M. Abadi, "Secrecy by Typing in Security Protocols," *Journal of the ACM*, vol. 46, no. 5, pp. 749–786, 1999.

[129] M. Backes, M. Maffei, and K. Pecina, "Automated Synthesis of Privacy-Preserving Distributed Applications," in *Proc. 19th Network and Distributed System Security Symposium (NDSS).* Internet Society, 2012.

[130] M. Maffei, K. Pecina, and M. Reinert, "Security and Privacy by Declarative Design," in *Proc. 26th IEEE Symposium on Computer Security Foundations (CSF).* IEEE Computer Society Press, 2013, p. 8196.

[131] P. Baillot and M. Hofmann, “Type Inference in Intuitionistic Linear Logic,” in *Proc. 12th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP).* ACM Press, 2010, pp. 219–230.

Part V

Appendix

A
Proofs of Chapter 2

A.1 Soundness of AF7

We present a complete soundness proof for our AF7 type system. The structure of the proof is standard: we first establish a Subject Reduction theorem, which shows that types are preserved upon reduction, and then we prove that well-typed programs are *statically* safe. By combining these two guarantees, we establish that the type system enforces our safety notion. Finally, we prove an Opponent Typability lemma, which states that any opponent is trivially well-typed: this allows us to carry out a simple proof of robust safety, based on our safety theorem.

The present appendix is organized as follows:

- Appendix A.1.1 develops basic properties of affine logic, which are needed in the soundness proof of the AF7 type system;

- Appendix A.1.2 establishes some basic results about AF7 and the environment rewriting relation;

- Appendix A.1.3 presents the main properties of kinding and subtyping, most notably the transitivity of the subtyping relation;

- Appendix A.1.4 establishes a standard substitution lemma;

- Appendix A.1.5 provides the inversion lemmas for the constructed values of our framework;

- Appendix A.1.6 presents the fundamental properties of the extraction relation;

- Appendix A.1.7 details the proof of the Subject Reduction theorem, building upon the results of the previous sections;

- Appendix A.1.8 presents the proof of robust safety.

A.1.1 Properties of the logic

We first show that affine logic is closed under substitution of variables with closed terms. This is important to prove the substitution lemma of our type system.

Lemma A.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. □

In the next result we recall that we write $\Delta \vdash \Delta'$ to stand for $\Delta \vdash F_1 \otimes \ldots \otimes F_n$ whenever $\Delta' = F_1, \ldots, F_n$. If Δ' is empty, we let $\Delta \vdash \Delta'$ stand for $\Delta \vdash \mathbf{1}$.

Lemma A.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:

$$\dfrac{\textsc{Ident}\,\dfrac{}{F_1 \vdash F_1} \qquad \dfrac{}{F_2 \vdash F_2}\,\textsc{Ident}}{F_1, F_2 \vdash F_1 \otimes F_2}\,\otimes\text{-}\textsc{Right}$$

     Since $F_1, F_2 \vdash F_1 \otimes F_2$ and $\Delta, F_1 \otimes F_2 \vdash F$ we can apply a Cut elimination argument 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.

□

The next result is a generalization to multisets of formulas of the standard Cut rule, characteristic of sequent calculi presentation of formal logic.

Lemma A.3 (Multicut)**.** *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 A.2 we also know that $F_1 \otimes \ldots \otimes F_n, \Delta'' \vdash \Delta'''$. We can conclude by a standard Cut elimination argument. □

The next technical lemma formalizes the intuition that exponential formulas can be proved an arbitrary number of times.

Lemma A.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 A.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 A.3.

□

A.1.2 Basic results

The next results are completely standard. In the following we typically let $\mathcal{J}$ range over the judgements $\{\diamond, T, F, T :: k, T <: U, E : T\}$.

Lemma A.5 (Derived Judgements). *It holds that:*

1. *If* $\Gamma; \Delta \vdash \diamond$, *then* $\mathit{fnfv}(\Delta) \subseteq \mathit{dom}(\Gamma)$ *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* $\mathit{fnfv}(T) \subseteq \mathit{dom}(\Gamma)$.
5. *If* $\Gamma; \Delta \vdash F$, *then* $\Gamma; \Delta \vdash \diamond$ *and* $\mathit{fnfv}(F) \subseteq \mathit{dom}(\Gamma)$.
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* $\mathit{fnfv}(E) \subseteq \mathit{dom}(\Gamma)$.

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

Lemma A.6 (Joining Envs). *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 B.3 (point 1). □

Notation A.1 (Environment Entry η). *We define an environment entry η to be either a type environment entry μ or a formula F.*

Notation A.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 A.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 $(\Gamma; \Delta) \bullet (\Gamma'; \Delta') \vdash \mathcal{J}$. □

The next lemma establishes some basic properties of the rewriting relation. Intuitively, we show that this relation is consistent with logical entailment, in that it satisfies some expected properties which hold true for the latter.

Lemma A.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 B.3. By Lemma A.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 A.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 A.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 B.3, we get $\Gamma; !\Delta', !\Delta' \vdash \diamond$ by multiple applications of (Form Env Entry). By Lemma A.4 we know that $\Delta \vdash !\Delta'$ implies $\Delta \vdash !\Delta', !\Delta'$, hence we conclude $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta', !\Delta'$ by using (Rewrite).

□

The next result states that, if an environment $\Gamma; \Delta$ can be rewritten to an environment $\Gamma; \Delta'$, then it can derive all the judgements provable by the latter.

Lemma A.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 A.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 A.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.

□

The next technical lemma states that rewriting does not introduce free variables.

Lemma A.10 (Rewriting and Variables). *If $x \notin dom(\Gamma)$ and $\Gamma; \Delta \hookrightarrow \Gamma; \Delta'$, then $x \notin fv(\Delta')$.*

Proof. Immediate by Lemma B.3 (point 1), since $\Gamma; \Delta \hookrightarrow \Gamma; \Delta'$ implies $\Gamma; \Delta' \vdash \diamond$ by inverting rule (Rewrite). □

The next lemma is an expected property of the refinement stripping function ψ, i.e., that it removes all the refinement formulas from a type.

Lemma A.11 (Soundness of ψ). *For every type T, we have $forms(x : \psi(T)) = \emptyset$.*

Proof. By induction on the structure of T. □

The next lemma states that the stripping function ψ is idempotent, i.e., there is no purpose in stripping refinements twice from the same type.

Lemma A.12 (Idempotent ψ). *For every type T, we have $\psi(\psi(T)) = \psi(T)$.*

Proof. By induction on the structure of T. □

A.1.3 Properties of kinding and subtyping

The next result states that, whenever a typing environment can assign a kind to a type T, then it can be rewritten so as to be split in two distinct components: the first one is exponential and it is needed to kind-check the structural information $\psi(T)$, while the second one can be used to derive the refinement formulas $forms(x : T)$ when T is tainted. This result is extensively used in the proofs, most likely to deal with the subtleties introduced by environment splitting.

Lemma A.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 $\Delta'' \vdash forms(x : T)$ for any $x \notin dom(\Gamma)$.*

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 B.3 (point 1), hence $\Gamma; \emptyset \vdash \alpha :: k$ by (Kind Var). Since $\Gamma; \Delta \hookrightarrow \Gamma; \emptyset$ by Lemma A.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 B.3 (point 1), hence $\Gamma; \emptyset \vdash \mathsf{unit} :: k$ by (Kind Unit). Since $\Gamma; \Delta \hookrightarrow \Gamma; \emptyset$ by Lemma A.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 A.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 A.8, we can conclude.

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

The next technical lemma is needed in the proof of Lemma B.14 below. It states that the assignment of a public kind does not depend on the refinement formulas associated to the type, but only on structural information.

Lemma A.14 (Bare Kinds Reverse)**.** *If* $\Gamma; \Delta \vdash T$ *and* $\Gamma; \Delta \vdash \psi(T) :: \mathsf{pub}$*, 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). □

The next result is similar in spirit to Lemma A.13, but it applies to subtyping. Again the goal is to identify a possible rewriting of the typing environment such that the structural subtyping relation and the refinement formulas can be proved separately. This is needed in a number of places to deal with the complications introduced by environment splitting.

Lemma A.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 B.3 (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 A.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 A.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 A.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 A.8, which implies $\Gamma; \Delta \hookrightarrow \Gamma; (!\Delta_{11}, !\Delta_{21}), \Delta_{22}$ again by Lemma A.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 A.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 A.8.

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

The next technical lemma is needed in the proof of Lemma B.14 below. It states that only refinement formulas are relevant for many judgements of our type system, so we can always replace a purely structural type $\psi(T)$ with any other (well-formed) purely structural type $\psi(T')$ in the typing environment.

Lemma A.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 B.3 when needed. □

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

The next technical lemma is needed in the proof of Lemma B.14 below. Intuitively, it states that kinding annotations for type variables do not play any role for many judgements of our type system.

Lemma A.17 (Type Variables and Kinding)**.** *For all* $\Gamma = \Gamma_0[(\alpha_i)^{i\in\{1,\ldots,n\}}]$ *and* $\hat{\Gamma} = \Gamma_0[(\alpha_i :: k_i)^{i\in\{1,\ldots,n\}}]$ *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 $dom(\alpha_i) = dom(\alpha_i :: k_i)$ 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 $dom(\alpha_i) = \{\alpha_i\} = dom(\alpha_i :: k_i)$.
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.

□

The next lemma states that any subtype of a public type is public, while any supertype of a tainted type is tainted. This is needed to prove Lemma B.16 below.

Lemma A.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 B.3 and Lemma A.17. Notice in particular that, by using Lemma B.3 and Lemma A.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 B.8.
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 B.8.

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 A.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 A.9. We apply Lemma A.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 A.8 we know that $\Gamma; \Delta, \Delta' \hookrightarrow \Gamma; \Delta_2$. We conclude by an application of Lemma A.9 that $\Gamma; \Delta, \Delta' \vdash T' :: \mathsf{tnt}$. Using Lemma A.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 B.3 we know that $\Gamma; \emptyset \vdash T$ and $\Gamma; \emptyset \vdash T'$ and thus by Lemma A.17 $\hat{\Gamma}; \emptyset \vdash T$ and $\hat{\Gamma}; \emptyset \vdash T'$.

1. By Lemma A.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 A.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 A.8 in combination with Lemma A.17, hence we conclude $\hat{\Gamma}; \Delta, \Delta' \vdash T :: \mathsf{pub}$ by Lemma A.9.

2. By Lemma A.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 A.8 in combination with Lemma A.17, hence we conclude $\hat{\Gamma}; \Delta, \Delta' \vdash T :: \mathsf{tnt}$ by Lemma A.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 A.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 A.8 in combination with Lemma A.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 A.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 A.8 and Lemma A.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 A.8 and Lemma A.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 A.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 A.8 and Lemma A.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 A.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 A.8 and Lemma A.17.

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

□

The next result is central to proving the transitivity of our subtyping relation. It establishes a standard characterization of public and tainted kinds: a type is public iff it is a subtype of Un, while it is tainted iff it is a supertype of Un.

Lemma A.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 A.8. The reverse implication follows immediately by Lemma B.14. □

The next technical lemma details a relationship between the stripping function ψ and the subtyping relation. It is invoked only once in the proof of transitivity for the subtyping relation.

Lemma A.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 B.3 (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 A.12. Furthermore, we know that $\mathit{forms}(y : \psi(T)) = \emptyset$ by Lemma A.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 B.8.
    - (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 A.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 A.12 we know that $\psi(\psi(T)) = \psi(T)$ and by Lemma A.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 B.8 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).

□

We are finally ready to prove the transitivity of the subtyping relation. This is a standard formulation for an affine setting.

Lemma A.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 B.3 it must be the case that $\Gamma; \Delta \vdash \diamond$ and $\Gamma; \Delta' \vdash \diamond$ and thus by Lemma A.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 B.8 to $\Gamma; \Delta' \vdash T' <: T''$.

Case $R_2 =$ (Sub Refl): Since in this case $T' = T''$, we can immediately conclude by applying Lemma B.8 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 B.14 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 A.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 B.14 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 A.20 it follows that:

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

By Lemma A.8 we know that $\Gamma; \Delta, \Delta' \hookrightarrow \Gamma; \Delta_1, \Delta'$, allowing us to conclude by an application of Lemma A.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 A.3 we can derive that $\Delta_2, \Delta'_2, \mathit{forms}(y : T) \vdash \mathit{forms}(y : T'')$. By applying rule (DERIVE) and Lemma A.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 A.8 and $\Gamma; \Delta_1, \Delta'_1 \vdash \psi(T) <: \psi(T'') = T <: T''$, we conclude $\Gamma; \Delta, \Delta' \vdash T <: T''$ by Lemma A.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 A.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 A.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 A.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. □

A.1.4 Properties of substitution

The next result establishes for value typing judgements a property we already showed for kinding and subtyping judgements. Namely, if a typing environment $\Gamma;\Delta$ can assign a type T to a value M, then it can be rewritten into two distinct typing environments: an exponential environment $\Gamma;!\Delta'$ where M is assigned the structural type $\psi(T)$ and a possibly non-exponential environment $\Gamma;\Delta''$ where the refinement formulas $\mathit{forms}(x:T)$ can be proved on the value M.

Lemma A.22 (Bare Types)**.** *Let $\mathit{fv}(M)=\emptyset$. 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 $\Delta''\vdash\mathit{forms}(x:T)\{M/x\}$ for any $x\notin\mathit{dom}(\Gamma)$.*

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 B.3, hence $\mathit{forms}(x:T)=\emptyset$ by Lemma A.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 B.3, hence $\Gamma;\emptyset\vdash y:T$ by (Val Var). This leads to the desired conclusion, since $\Gamma;\Delta\hookrightarrow\Gamma;\emptyset$ by Lemma A.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 $\mathit{forms}(x:\mathsf{unit})=\emptyset$, we just need to show that $\Gamma;\Delta\hookrightarrow\Gamma;!\Delta'$ for some $!\Delta'$ such that $\Gamma;!\Delta'\vdash():\mathsf{unit}$. By Lemma B.3 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 A.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 $\mathit{forms}(x:(y:U_1\to U_2))=\emptyset$, 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 $\Delta_{12}\vdash\mathit{forms}(x:U)\{M/x\}$. 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 $\Delta_{12},\Delta_2\vdash\mathit{forms}(x:U)\{M/x\}\otimes F\{M/x\}$, which is equivalent to $\Delta_{12},\Delta_2\vdash\mathit{forms}(x:\{x:U\mid F\})\{M/x\}$. Since $\Gamma;\Delta\hookrightarrow\Gamma;!\Delta_{11},(\Delta_{12},\Delta_2)$ by Lemma A.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 $\Delta_{12}\vdash\mathit{forms}(x:U)\{M/x\}$. By the premise $\Gamma;\Delta_2\vdash U<:T$ and Lemma A.15, we have $\Gamma;\Delta_2\hookrightarrow\Gamma;!\Delta_{21},\Delta_{22}$ with $\Gamma;!\Delta_{21}\vdash\psi(U)<:\psi(T)$ and $\Delta_{22},\mathit{forms}(x:U)\vdash\mathit{forms}(x:T)$. Now we note that

$x \notin \mathit{dom}(\Gamma)$ implies $x \notin \mathit{fv}(\Delta_{22})$ by Lemma A.10. Given that logical entailment is closed under substitution of closed values for variables by Lemma A.1, we get:

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

hence by Lemma A.3 we have:

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

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 A.8.

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

The next lemma states that multiplicative conjunctions occurring in refinement types can be equivalently broken into their atomic components. This is needed in the proof of Lemma A.24 below.

Lemma A.23 ($\otimes$ Sub)**.** *If* $\Gamma; \Delta \vdash \{x : T \mid F_1 \otimes F_2\}$*, then* $\Gamma; \emptyset \vdash \{x : T \mid F_1 \otimes F_2\} <:> \{x : \{x : T \mid F_1\} \mid F_2\}$*.*

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

The next result is a very convenient lemma, which is needed in the proof of our substitution lemma. It essentially states that, if a typing environment $\Gamma; \Delta$ can assign a type T to a value M, then it can be rewritten into two distinct typing environments: an exponential environment $\Gamma; !\Delta'$ where M is assigned the structural type $\psi(T)$ and a possibly non-exponential environment $\Gamma; \Delta''$ where M is assigned the original type T. Hence, purely structural typing judgements can be proved arbitrarily many times. This is again needed to deal with the subtleties introduced by environment splitting.

Lemma A.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 A.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 A.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 A.23. Hence, we proved $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta', (!\Delta', \Delta'')$ with $\Gamma; !\Delta' \vdash M : \psi(T)$ and $\Gamma; !\Delta', \Delta'' \vdash M : T$. □

The next simple lemma states that, if a value M is assigned a refinement type T, then the refinement formulas $\mathit{forms}(x : T)$ can be proved on M from the formulas in the typing environment.

Lemma A.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 A.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 A.3. □

The next lemma establishes some basic syntactic properties of substitution.

Lemma A.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. □

Finally, we can state and prove our substitution lemma, showing that typing is preserved by substitution of closed values for variables with the same type. The statement is complicated by the necessity to join different environments, but the formulation is consistent with standard presentations of substructural type systems.

Lemma A.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 B.3 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 B.3, 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 A.1, we have:

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

   Now we note that by Lemma A.25 we have $\Delta \vdash \mathit{forms}(x : U)\{M/x\}$, hence $\Delta, (\Delta', \Delta'')\{M/x\} \vdash F\{M/x\}$ by Lemma A.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 A.1, we have:

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

   Now we note that by Lemma A.25 we have $\Delta \vdash \mathit{forms}(x : U)\{M/x\}$, hence $\Delta, (\Delta', \Delta'')\{M/x\} \vdash \Delta^*\{M/x\}$ by Lemma A.3.

   By Lemma A.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 B.3 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 B.3 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 A.24 in combination with Lemma A.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 A.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 A.24 in combination with Lemma A.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 A.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.

□

A.1.5 Inversion lemmas

The next result is a standard bound weakening lemma: any occurrence of a type T in the typing environment can be safely replaced with a subtype T'.

Lemma A.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 B.8 implicitly.

1. $\mathcal{J} = \diamond$: The induction proof uses the fact that by Lemma B.3 we know that $\mathit{fnfv}(T') \subseteq \mathit{dom}(\Gamma)$ and $\mathit{fnfv}(\Delta) \subseteq \mathit{dom}(\Gamma)$.

2. $\mathcal{J} = U$: By Lemma B.3 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 B.3, and the fact that $\mathit{dom}(\Gamma, x : \psi(T), \Gamma') = \mathit{dom}(\Gamma, x : \psi(T'), \Gamma')$. Furthermore, it applies Lemma A.15 to $\Gamma; \Delta \vdash T' <: T$ (showing that formulas in Δ and T' entail those in T) in combination with Lemma A.3 to conclude.

4. $\mathcal{J} = U :: k$: The proof makes uses of the previous statements. It also uses Lemma A.16 to show that replacing $x : \psi(T)$ by $x : \psi(T')$ is safe. It applies Lemma A.15 to $\Gamma; \Delta \vdash T' <: T$ (showing that formulas in Δ and T' entail those in T) in combination with Lemma A.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 A.15 and Lemma A.3.

□

We now present two technical lemmas which are needed to establish the inversion result for iso-recursive type constructors.

Lemma A.29 (Type Variables and Kinding)**.** *If $\Gamma, \alpha, \Gamma'; \Delta \vdash T :: k$, then $\alpha \notin$ fnfv(T).*

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

The case (Kind Unit) follows immediately, since *fnfv*(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 B.3 and having both α and $\alpha :: k$ in the same environment would violate the well-formedness conditions enforced by (Type Env Entry), given that *dom*$(\alpha) =$ *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 $\alpha \in$ *fnfv*(T) if and only if $\alpha \in$ *fnfv*$(\psi(T))$. The remaining cases follow by the induction hypothesis. □

Lemma A.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 B.3 and Lemma A.6.

2. By induction on the derivation of $\Gamma, \alpha :: k, \Gamma'; \Delta \vdash \diamond$, making use of Lemma B.3 and Lemma A.6.

3. By the definition of (TYPE), using point (2), Lemma B.3 and Lemma A.6.

4. Since we know that $T = \psi(T)$, we can apply Lemma A.13 in combination with Lemma A.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 A.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 A.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 A.15 in combination with Lemma A.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 A.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 A.9.

□

We can finally state and prove a number of inversion results for the constructed values of our framework. The goal is showing that the elementary components of these constructed values have indeed the expected types. There is a substantial amount of work to do, but the technical details are mostly standard.

Lemma A.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 B.3, we immediately derive $\Gamma;\emptyset \vdash x : T \to U <: \psi(V)$ by (Sub Refl). Since $\Gamma;\Delta \hookrightarrow \Gamma;\Delta$ by Lemma A.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 A.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 A.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 A.8 point 1, we have $\Gamma; \Delta_2 \vdash \psi(V') <: \psi(V)$ by Lemma A.9. By transitivity of the subtyping relation (Lemma B.17) 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 B.3 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 B.8 to extend Γ with $x : \psi(T')$ in the second judgement, we can conclude, since $\Gamma; \Delta \hookrightarrow \Gamma; \emptyset$ by Lemma A.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 A.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 A.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 B.10 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 A.8, we can conclude.

4. Follows immediately from statement (3) by an application of Lemma A.9.

□

Lemma A.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 A.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 A.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 B.8 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 B.12 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 A.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 A.8 we know that:

$$\Gamma; \Delta \hookrightarrow \Gamma; (!\Delta_{11}, !\Delta_{21}), (!\Delta_{12}, !\Delta_{11}, !\Delta_{22}),$$

which allows us to conclude.

□

Lemma A.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*
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 A.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 A.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 B.3 we know that:

$$\Gamma; !\Delta_{22} \vdash U.$$

Using Lemma B.8 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 A.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 A.9.

6. The statement follows immediately by an application of statement (4) and Lemma A.9.

□

Lemma A.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 A.31, part (1).
2. By induction on the derivation of $\Gamma;\Delta \vdash \mu\alpha.T <: \mu\alpha.T'$. We implicitly use Lemma B.3 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 A.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 A.8, we can apply (Sub Pos Rec) to derive that:

$$\Gamma;!\Delta' \vdash \mu\alpha.T <: \mu\alpha.T'.$$

By point (6) of Lemma B.13, 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 A.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 A.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 B.13 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 A.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 A.8, we can conclude.

4. We can immediately conclude by an application of statement (3) and Lemma A.9.

□

A.1.6 Properties of extraction

We first present some simple, but useful properties of the extraction relation.

Lemma A.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 A.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 A.37 (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 $\mathit{fn}(F) \cap \{\widetilde{a}\} = \emptyset$: we have $E \rightsquigarrow^{\widetilde{a}} [F \mid \mathsf{assume}\ \mathbf{1}]$ by (Extr Assume). Since $\{\widetilde{b}\} \subseteq \{\widetilde{a}\}$, we know that $\mathit{fn}(F) \cap \{\widetilde{b}\} = \emptyset$, 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 $\mathit{fn}(F) \cap \{\widetilde{a}\} \neq \emptyset$: we have $E \rightsquigarrow^{\widetilde{a}} [\emptyset \mid \mathsf{assume}\ F]$ by (Extr Exp). Now we distinguish two cases: if $\mathit{fn}(F) \cap \{\widetilde{b}\} \neq \emptyset$, 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 $\mathit{fn}(F) \cap \{\widetilde{b}\} = \emptyset$, 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 $E = E_1 \Rsh E_2$: we know by the definition of the only applicable extraction rule (Extr Fork) that:

- $E \rightsquigarrow^{\widetilde{a}} [\Delta_1, \Delta_2 \mid D_1 \Rsh D_2]$ and
- $E \rightsquigarrow^{\widetilde{b}} [\Delta'_1, \Delta'_2 \mid D'_1 \Rsh D'_2]$, 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:

$$D_1 \Rsh D_2 \rightsquigarrow^{\widetilde{b}} [\Delta''_1, \Delta''_2 \mid D'_1 \Rsh D'_2],$$

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 $E \rightsquigarrow^{\widetilde{a}} [\Delta \mid D]$ and $E \rightsquigarrow^{\widetilde{b}} [\Delta' \mid D']$ 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 $E \rightsquigarrow^{\widetilde{a}} [\emptyset \mid E]$ and $E \rightsquigarrow^{\widetilde{b}} [\emptyset \mid E]$ by (Extr Exp), so we immediately conclude.

□

Lemma A.38 (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 A.39 (Idempotent Extraction). *If $E \rightsquigarrow^{\widetilde{a}} [\Delta \mid D]$, then $D \rightsquigarrow^{\widetilde{a}} [\emptyset \mid D]$.*

Proof. By induction on the derivation of $E \rightsquigarrow^{\widetilde{a}} [\Delta \mid D]$. □

The next result shows that heating preserves logic: if $E \Rrightarrow E'$, then the formulas extracted from E are exactly the same of the formulas extracted from E'. Moreover, the purged expressions D and D' obtained after extracting the assumptions from E and E' respectively are again related by heating. All this information is needed to show that heating preserves typing (Lemma A.45 below). In the following proofs we often write $E \rightsquigarrow [\Delta \mid D]$ whenever $E \rightsquigarrow^{\widetilde{a}} [\Delta \mid D]$ for some $\widetilde{a}$ clear from the context.

Lemma A.40 (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 $a \notin fn(E)$. 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 $a \notin fn(E)$, we can apply Lemma A.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 $a \notin fn(E)$ implies $a \notin fn(D)$ by Lemma A.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:

$$\dfrac{\textsc{Extr Let}\ \dfrac{E \rightsquigarrow^{a,\widetilde{b}} [\Delta \mid D]}{\mathsf{let}\ x = E\ \mathsf{in}\ E' \rightsquigarrow^{a,\widetilde{b}} [\Delta \mid \mathsf{let}\ x = D\ \mathsf{in}\ E']}}{(\nu a)(\mathsf{let}\ x = E\ \mathsf{in}\ E') \rightsquigarrow^{\widetilde{b}} [\Delta \mid (\nu a)(\mathsf{let}\ x = D\ \mathsf{in}\ E')]}\ \textsc{Extr Res}$$

Since $a \notin \mathit{fn}(E')$ implies $a \notin \mathit{fn}(D)$ by Lemma A.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:

$$\textsc{Exp Fork}\ \dfrac{\textsc{Exp Fork}\ \dfrac{E \rightsquigarrow [\Delta \mid D] \qquad E' \rightsquigarrow [\Delta' \mid D']}{E \Rsh E' \rightsquigarrow [\Delta, \Delta' \mid D \Rsh D']} \qquad E'' \rightsquigarrow [\Delta'' \mid D'']}{(E \Rsh E') \Rsh E'' \rightsquigarrow [\Delta, \Delta', \Delta'' \mid (D \Rsh D') \Rsh D'']}$$

Hence, we can construct the following derivation:

$$\textsc{Exp Fork}\ \dfrac{E \rightsquigarrow [\Delta \mid D] \qquad \dfrac{E' \rightsquigarrow [\Delta' \mid D'] \qquad E'' \rightsquigarrow [\Delta'' \mid D'']}{E' \Rsh E'' \rightsquigarrow [\Delta', \Delta'' \mid D' \Rsh D'']}\ \textsc{Exp Fork}}{E \Rsh (E' \Rsh E'') \rightsquigarrow [\Delta, \Delta', \Delta'' \mid D \Rsh (D' \Rsh D'')]}$$

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:

$$\textsc{Exp Fork}\ \dfrac{\textsc{Exp Fork}\ \dfrac{E \rightsquigarrow [\Delta \mid D] \qquad E' \rightsquigarrow [\Delta' \mid D']}{E \Rsh E' \rightsquigarrow [\Delta, \Delta' \mid D \Rsh D']} \qquad E'' \rightsquigarrow [\Delta'' \mid D'']}{(E \Rsh E') \Rsh E'' \rightsquigarrow [\Delta, \Delta', \Delta'' \mid (D \Rsh D') \Rsh D'']}$$

Hence, we can construct the following derivation:

$$\textsc{Exp Fork}\ \dfrac{\textsc{Exp Fork}\ \dfrac{E' \rightsquigarrow [\Delta' \mid D'] \qquad E \rightsquigarrow [\Delta \mid D]}{E' \Rsh E \rightsquigarrow [\Delta, \Delta' \mid D' \Rsh D]} \qquad E'' \rightsquigarrow [\Delta'' \mid D'']}{(E' \Rsh E) \Rsh E'' \rightsquigarrow [\Delta, \Delta', \Delta'' \mid (D' \Rsh D) \Rsh D'']}$$

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 $\mathsf{let}\ x = (E_1 \Rsh E_2)\ \mathsf{in}\ E_3 \rightsquigarrow [\Delta_1, \Delta_2 \mid \mathsf{let}\ x = (D_1 \Rsh D_2)\ \mathsf{in}\ E_3]$ with $E_1 \rightsquigarrow [\Delta_1 \mid D_1]$ and $E_2 \rightsquigarrow [\Delta_2 \mid D_2]$. In fact, the only possible extraction derivation is the following:

$$\textsc{Extr Let}\ \frac{\textsc{Extr Fork}\ \dfrac{E_1 \rightsquigarrow [\Delta_1 \mid D_1] \qquad E_2 \rightsquigarrow [\Delta_2 \mid D_2]}{E_1 \Rsh E_2 \rightsquigarrow [\Delta_1, \Delta_2 \mid D_1 \Rsh D_2]}}{\mathsf{let}\ x = (E_1 \Rsh E_2)\ \mathsf{in}\ E_3 \rightsquigarrow [\Delta_1, \Delta_2 \mid \mathsf{let}\ x = (D_1 \Rsh D_2)\ \mathsf{in}\ E_3]}$$

Hence, we can construct the following derivation:

$$\textsc{Extr Fork}\ \frac{E_1 \rightsquigarrow [\Delta_1 \mid D_1] \qquad \dfrac{E_2 \rightsquigarrow [\Delta_2 \mid D_2]}{\mathsf{let}\ x = E_2\ \mathsf{in}\ E_3 \rightsquigarrow [\Delta_2 \mid \mathsf{let}\ x = D_2\ \mathsf{in}\ E_3]}\ \textsc{Extr Let}}{E_1 \Rsh (\mathsf{let}\ x = E_2\ \mathsf{in}\ E_3) \rightsquigarrow [\Delta_1, \Delta_2 \mid D_1 \Rsh (\mathsf{let}\ x = D_2\ \mathsf{in}\ E_3)]}$$

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.

□

The next lemma is in the same spirit of Lemma A.40, but it predicates over the reduction relation rather than on heating and it is slightly more complicated. This is needed in the proof of Subject Reduction (Theorem A.1 below).

Lemma A.41 (Reduction Preserves Logic). *If $E \to E'$ and $E \rightsquigarrow^{\widetilde{a}} [\Delta \mid D]$, then $D \to D'$ and $E' \rightsquigarrow^{\widetilde{a}} [\Delta, \Delta' \mid D'']$ for some D', D'', Δ' such that $D' \rightsquigarrow^{\widetilde{a}} [\Delta' \mid D^*]$ 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 $E \rightsquigarrow [\emptyset \mid E]$, 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 $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 \rightsquigarrow [\Delta_1 \mid \mathsf{let}\ x = D_1\ \mathsf{in}\ E_2]$ with $E_1 \rightsquigarrow [\Delta_1 \mid D_1]$. 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 $\mathsf{let}\ x = D_1\ \mathsf{in}\ E_2 \to \mathsf{let}\ x = D_1'\ \mathsf{in}\ E_2$ by (Red Let). Now we observe that $\mathsf{let}\ x = E_1'\ \mathsf{in}\ E_2 \rightsquigarrow [\Delta_1, \Delta_1' \mid \mathsf{let}\ x = D'\ \mathsf{in}\ E_2]$ and $\mathsf{let}\ x = D_1'\ \mathsf{in}\ E_2 \rightsquigarrow [\Delta_1' \mid \mathsf{let}\ x = D_1''\ \mathsf{in}\ E_2]$, so we conclude by (Heat Let).

Case (Red Res): assume $(\nu a)E \to (\nu a)E'$ with $E \to E'$ and $(\nu a)E \rightsquigarrow^{\widetilde{b}} [\Delta_1 \mid (\nu a)D_1]$ with $E \rightsquigarrow^{a,\widetilde{b}} [\Delta_1 \mid D_1]$. By induction hypothesis $D_1 \to D_1'$ and $E' \rightsquigarrow^{a,\widetilde{b}} [\Delta_1, \Delta_1' \mid D']$ with $D_1' \rightsquigarrow^{a,\widetilde{b}} [\Delta_1' \mid D_1'']$ and $D_1'' \Rrightarrow D'$. We then have $(\nu a)D_1 \to (\nu a)D_1'$ by (Red Res). Now we observe that $(\nu a)E' \rightsquigarrow^{\widetilde{b}} [\Delta_1, \Delta_1' \mid (\nu a)D']$ and $(\nu a)D_1' \rightsquigarrow^{\widetilde{b}} [\Delta_1' \mid (\nu a)D_1'']$, so we conclude by (Heat Res).

Case (Red Fork 1): assume $E_1 \Rsh E_2 \rightarrow E_1' \Rsh E_2$ with $E_1 \rightarrow E_1'$ and $E_1 \Rsh E_2 \rightsquigarrow [\Delta_1, \Delta_2 \mid D_1 \Rsh D_2]$ with $E_1 \rightsquigarrow [\Delta_1 \mid D_1]$, $E_2 \rightsquigarrow [\Delta_2 \mid D_2]$. By induction hypothesis $D_1 \rightarrow D_1'$ 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 $D_1 \Rsh D_2 \rightarrow D_1' \Rsh D_2$ by (Red Fork 1). Now we observe that $E_1' \Rsh E_2 \rightsquigarrow [\Delta_1, \Delta_1', \Delta_2 \mid D' \Rsh D_2]$ and $D_1' \Rsh D_2 \rightsquigarrow [\Delta_1' \mid D_1'' \Rsh D_2]$, since $D_2 \rightsquigarrow [\emptyset \mid D_2]$ by Lemma A.39. Thus, we conclude by (Heat Fork 1).

Case (Red Fork 2): analogous to the previous case.

Case (Red Heat): assume $E \rightarrow E'$ by the premises $E \Rrightarrow E_A$, $E_A \rightarrow E_B$, $E_B \Rrightarrow E'$. Assume further $E \rightsquigarrow [\Delta_1 \mid E_1]$. By Lemma A.40 we have $E_A \rightsquigarrow [\Delta_1 \mid E_A']$ with $E_1 \Rrightarrow E_A'$. By inductive hypothesis we get $E_A' \rightarrow E_B'$ 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 A.40 we have $E' \rightsquigarrow [\Delta_1, \Delta_1' \mid E'']$ with $D_B \Rrightarrow E''$. Since we can derive $E_1 \rightarrow E_B'$ by (Red Heat) and $E_B'' \Rrightarrow E''$ by (Heat Trans), we can conclude.

□

A.1.7 Proof of subject reduction

In the proof of Lemmas A.43, A.44, A.45 and Theorem A.1 below we rely on an observation about the structure of the type derivations to simplify the formal reasoning and carry out the proofs. First, we consider an alternative formulation of typing for values, presented in Table A.1, which removes the non-structural rule (Val Refine). We also assume to keep the original typing rules for expressions.

We can show that the original and the alternative formulation coincide.

Lemma A.42 (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:

$$\dfrac{\textsc{Val Var Refine}\ \dfrac{(x : T) \in \Gamma \qquad \Gamma; \Delta \vdash \mathbf{1}}{\Gamma; \Delta \vdash^{\mathsf{alt}} x : \{y : T \mid \mathbf{1}\}} \qquad \dfrac{\Gamma; \emptyset \vdash \psi(T) <: \psi(T) \qquad \Gamma, y : \psi(T); \mathbf{1} \vdash \mathbf{1}}{\Gamma; \emptyset \vdash \{y : T \mid \mathbf{1}\} <: T}\ \textsc{Sub Refine}}{\Gamma; \Delta \vdash^{\mathsf{alt}} x : T}\ \textsc{Exp Subsum}$$

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:

$$\mathcal{R}\ \dfrac{(\ldots) \qquad \Gamma; \Delta^* \vdash F'\{M/x\} \qquad \Gamma; \Delta' \hookrightarrow \Gamma; (\ldots), \Delta^*}{\Gamma; \Delta' \vdash^{\mathsf{alt}} M : \{x : U \mid F'\}}$$

[Val Var Refine]
$$\frac{(x:T)\in\Gamma \qquad \Gamma;\Delta\vdash F\{x/y\}}{\Gamma;\Delta\vdash^{\mathsf{alt}} x:\{y:T\mid F\}}$$

[Val Unit Refine]
$$\frac{\Gamma;\Delta\vdash F\{()/y\}}{\Gamma;\Delta\vdash^{\mathsf{alt}} ():\{y:\mathsf{unit}\mid F\}}$$

[Val Fun Refine]
$$\frac{\begin{array}{c}(\Gamma;!\Delta_1)\bullet x:T\vdash^{\mathsf{alt}} E:U\\ \Gamma;\Delta_2\vdash F\{\lambda x.\,E/y\}\\ \Gamma;\Delta\hookrightarrow\Gamma;!\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta\vdash^{\mathsf{alt}} \lambda x.\,E:\{y:x:T\to U\mid F\}}$$

[Val Pair Refine]
$$\frac{\begin{array}{c}\Gamma;!\Delta_1\vdash^{\mathsf{alt}} M:T\\ \Gamma;!\Delta_2\vdash^{\mathsf{alt}} N:U\{M/x\}\\ \Gamma;\Delta_3\vdash F\{(M,N)/y\}\\ \Gamma;\Delta\hookrightarrow\Gamma;!\Delta_1,!\Delta_2,\Delta_3\end{array}}{\Gamma;\Delta\vdash^{\mathsf{alt}} (M,N):\{y:x:T*U\mid F\}}$$

[Val Inl Refine]
$$\frac{\begin{array}{c}\Gamma;!\Delta_1\vdash^{\mathsf{alt}} M:T \qquad \Gamma;!\Delta_1\vdash U\\ \Gamma;\Delta_2\vdash F\{\mathsf{inl}\ M/y\} \qquad \Gamma;\Delta\hookrightarrow\Gamma;!\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta\vdash^{\mathsf{alt}} \mathsf{inl}\ M:\{y:T+U\mid F\}}$$

[Val Inr Refine]
$$\frac{\begin{array}{c}\Gamma;!\Delta_1\vdash^{\mathsf{alt}} M:U \qquad \Gamma;!\Delta_1\vdash T\\ \Gamma;\Delta_2\vdash F\{\mathsf{inr}\ M/y\} \qquad \Gamma;\Delta\hookrightarrow\Gamma;!\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta\vdash^{\mathsf{alt}} \mathsf{inr}\ M:\{y:T+U\mid F\}}$$

[Val Fold Refine]
$$\frac{\begin{array}{c}\Gamma;!\Delta_1\vdash^{\mathsf{alt}} M:T\{\mu a.T/\alpha\}\\ \Gamma;\Delta_2\vdash F\{\mathsf{fold}\ M/y\} \qquad \Gamma;\Delta\hookrightarrow\Gamma;!\Delta_1,\Delta_2\end{array}}{\Gamma;\Delta\vdash^{\mathsf{alt}} \mathsf{fold}\ M:\{y:\mu\alpha.T\mid F\}}$$

Table A.1: Alternative rules for typing values (AF7)

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 B.17, and the environment rewriting relation, proved in Lemma A.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 A.15 in combination with Lemma A.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 A.8, we conclude $\Gamma;\Delta \vdash^{\mathsf{alt}} M : \{x : T \mid F\}$ by a variant of Lemma A.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 B.3, hence $\Gamma;\emptyset \vdash \diamond$ again by Lemma B.3 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.

□

Now the idea is to appeal to the transitivity of both the subtyping relation (Lemma B.17) and the environment rewriting relation (Lemma A.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:

$$\dfrac{\Gamma;\Delta_1 \vdash^{\mathsf{alt}} E : T_1 \qquad \Gamma;\Delta^* \vdash T_1 <: T \qquad \Gamma;\Delta \hookrightarrow \Gamma;\Delta_1,\Delta^*}{\Gamma;\Delta \vdash^{\mathsf{alt}} E : T}$$

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 A.42, we note that the previous transformation can be applied to any type derivation.

Now we can show that extraction preserves typing: this is needed to show that heating preserves typing (Lemma A.45 below).

Lemma A.43 (Extraction Preserves Typing). *If* $\Gamma; \Delta \vdash E : T$ *and* $E \rightsquigarrow^{\widetilde{a}} [\Delta' \mid E']$*, then* $\Gamma; \Delta, \Delta' \vdash E' : T$.

Proof. By a case analysis on the structure of E:

Case E is any expression such that $E \rightsquigarrow [\emptyset \mid E]$: the conclusion is trivial.

Case $E = \mathsf{assume}\ F$ with $F \neq \mathbf{1}$ and $\mathit{fn}(F) \cap \{\widetilde{a}\} = \emptyset$: we have $E \rightsquigarrow^{\widetilde{a}} [F \mid \mathsf{assume}\ \mathbf{1}]$ 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 $E \rightsquigarrow^{\widetilde{b}} [\Delta' \mid (\nu a)D']$ with $D \rightsquigarrow^{a,\widetilde{b}} [\Delta' \mid D']$ 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:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $D \rightsquigarrow^{a} [\Delta'' \mid D'']$
- $\Gamma, a \updownarrow T'; \Delta_A, \Delta'' \vdash D'' : V$

By Lemma A.37 we know that $D' \rightsquigarrow^{a} [\Delta''' \mid D'']$, for some Δ''' such that $\Delta'' = \Delta', \Delta'''$. We can then construct the following type derivation:

$$\dfrac{\textsc{Exp Res}\ \dfrac{D' \rightsquigarrow^{a} [\Delta''' \mid D''] \qquad \Gamma, a \updownarrow T'; \Delta_A, \overbrace{\Delta', \Delta'''}^{\Delta''} \vdash D'' : V}{\Gamma; \Delta_A, \Delta' \vdash (\nu a)D' : V} \qquad \Gamma; \Delta_B \vdash V <: T}{\Gamma; \Delta, \Delta' \vdash (\nu a)D' : T}\ \textsc{Exp Subsum}$$

Case $E = \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$: we have $E \rightsquigarrow^{\widetilde{a}} [\Delta' \mid \mathsf{let}\ x = D'\ \mathsf{in}\ E_2]$ with $E_1 \rightsquigarrow^{\widetilde{a}} [\Delta' \mid D']$ 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 A.37 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.

□

Similarly to the previous result, we can also show that *inverting* an extraction preserves typing: again, this is needed to prove that heating preserves typing (Lemma A.45 below).

Lemma A.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 A.38 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 B.8 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 A.9 and get:

$$\Gamma, a \updownarrow W; \Delta, \Delta', \Delta'' \vdash D'' : T.$$

Finally, we note that $a \notin fn(T)$ by applying Lemma B.3 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 $E \rightsquigarrow^{\tilde{b}} [\Delta' \mid \mathsf{let}\ x = D_1\ \mathsf{in}\ E_2]$, where $E_1 \rightsquigarrow^{\tilde{b}} [\Delta' \mid D_1]$ 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 A.38 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 B.8 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 A.8.

Case $E = E_1 \Rsh E_2$: similar to the previous case.

□

The next result, sometimes called Subject Heating, shows that typing is preserved by heating. This is needed in the proof of the Subject Reduction theorem, since the reduction relation is closed under heating.

Lemma A.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 A.40 we know that $E \Rrightarrow E'$ implies $E' \rightsquigarrow [\Delta' \mid D']$ with $D \Rrightarrow D'$. Since Lemma A.40 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 A.40 we know that $E \Rrightarrow E'$ implies $E' \rightsquigarrow^a [\Delta' \mid D']$ with $D \Rrightarrow D'$. Since Lemma A.40 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 $E \upharpoonright E'' \Rrightarrow E' \upharpoonright E''$ by the premise $E \Rrightarrow E'$. Assume further that $\Gamma; \Delta \vdash E \upharpoonright E'' : T$. The judgement must follow by an instance of (Exp Fork) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash E \upharpoonright E'' : V$ 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 A.40 we know that $E \Rrightarrow E'$ implies $E' \rightsquigarrow [\Delta' \mid D']$ with $D \Rrightarrow D'$. Since Lemma A.40 is depth-preserving, we can apply the inductive hypothesis and get $\Gamma; \Delta_1 \vdash D' : U$, hence the conclusion $\Gamma; \Delta \vdash E' \upharpoonleft E'' : T$ follows by applying (Exp Fork) and (Exp Subsum).

Case (Heat Fork 2): the case is analogous to Heat Fork 1.

Case (Heat Fork ()): assume $() \upharpoonleft E \Rrightarrow E$ with $\Gamma; \Delta \vdash () \upharpoonleft E : T$. The judgement must follow by an instance of (Exp Fork) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash () \upharpoonleft E : V$ 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 B.3, thus $\Gamma; \Delta_1, \Delta_2 \vdash \diamond$ by Lemma A.6. By Lemma B.8 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 A.9 and this implies $\Gamma; \Delta_A \vdash E : V$ by Lemma A.44. The conclusion $\Gamma; \Delta \vdash E : T$ follows by (Exp Subsum).

Assume now $E \Rrightarrow () \upharpoonleft E$ 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 B.3 we know that $\Gamma; \Delta_A \vdash E : V$ implies $\Gamma; \Delta_A \vdash \diamond$, hence $\Gamma; \emptyset \vdash \diamond$ again by Lemma B.3 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 A.43. 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 $\Gamma; \Delta_A \vdash () \upharpoonleft E : V$ by (Exp Fork). The conclusion $\Gamma; \Delta \vdash () \upharpoonleft E : T$ follows by (Exp Subsum).

Case (Heat Msg ()): let $a!M \Rrightarrow a!M \Rsh ()$ 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 B.3 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 B.3 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 $\Gamma;\Delta_A \vdash a!M \Rsh () : \mathsf{unit}$ by (Exp Fork). Hence, the conclusion $\Gamma;\Delta \vdash a!M \Rsh () : T$ 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:

$$\dfrac{\textsc{Exp Fork}\ \dfrac{\textsc{Exp True}\ \dfrac{\Gamma;\emptyset \vdash \diamond}{\Gamma;\emptyset \vdash \mathsf{assume}\ \mathbf{1} : \mathsf{unit}} \quad \textsc{Val Unit}\ \dfrac{\dfrac{\Gamma;\Delta_1 \vdash \diamond}{\Gamma;\Delta_1 \vdash () : \mathsf{unit}} \qquad \Gamma;\Delta_2 \vdash \mathsf{unit} <: V}{\Gamma;\Delta_A, F \vdash () : V}\ \textsc{Exp Subsum}}{\Gamma;\Delta_A \vdash \mathsf{assume}\ F \Rsh () : V} \qquad \Gamma;\Delta_B \vdash V <: T}{\Gamma;\Delta \vdash \mathsf{assume}\ F \Rsh () : T}\ \textsc{Exp Subsum}$$

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 $a \notin fn(E')$. 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 A.39, 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 A.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 A.39, 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 A.43. 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 A.39, 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:

$$\textsc{Exp Res}\ \dfrac{\textsc{Exp Let}\ \dfrac{D \rightsquigarrow^{\emptyset} [\Delta'' \mid D'] \qquad \dfrac{(1)}{\Gamma, a \updownarrow T'; \Delta_1, \Delta'' \vdash D' : U} \qquad \dfrac{(\Gamma;\Delta_2)\bullet x : U \vdash E' : V}{(\Gamma, a \updownarrow T';\Delta_2)\bullet x : U \vdash E' : V}}{\Gamma, a \updownarrow T'; \Delta_A, \Delta' \vdash \mathsf{let}\ x = D\ \mathsf{in}\ E' : V}}{\Gamma;\Delta_A \vdash (\nu a)(\mathsf{let}\ x = E\ \mathsf{in}\ E') : V}$$

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 $D \rightsquigarrow^{\emptyset} [\Delta'' \mid D']$ using Lemma A.43. We conclude $\Gamma;\Delta \vdash (\nu a)(\mathsf{let}\ x = E\ \mathsf{in}\ E') : T$ by (Exp Subsum).

Case (Heat Fork Comm): 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 A.39, 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 $E' \upharpoonright E \rightsquigarrow [\Delta_1, \Delta_2 \mid D_2 \upharpoonright D_1]$ by applying (Extr Fork) to $E \rightsquigarrow [\Delta_1 \mid D_1]$ and $E' \rightsquigarrow [\Delta_2 \mid D_2]$, 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 A.8. Finally, we conclude $\Gamma; \Delta \vdash (E' \upharpoonright E) \upharpoonright E'' : T$ by (Exp Subsum).

Case (Heat Fork Assoc): 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:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $E \upharpoonright E' \rightsquigarrow [\Delta_1, \Delta_2 \mid D_1 \upharpoonright D_2]$ with $E \rightsquigarrow [\Delta_1 \mid D_1]$ and $E' \rightsquigarrow [\Delta_2 \mid D_2]$
- $E'' \rightsquigarrow [\Delta_3 \mid D_3]$
- $\Gamma; \Delta_A, \Delta_1, \Delta_2, \Delta_3 \hookrightarrow \Gamma; \Delta'_1, \Delta'_2$
- $\Gamma; \Delta'_1 \vdash D_1 \upharpoonright D_2 : U$
- $\Gamma; \Delta'_2 \vdash D_3 : V$

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 $D_1 \rightsquigarrow [\emptyset \mid D_1]$ and $D_2 \rightsquigarrow [\emptyset \mid D_2]$ by Lemma A.39, 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 $E' \upharpoonright E'' \rightsquigarrow [\Delta_2, \Delta_3 \mid D_2 \upharpoonright D_3]$ 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 A.39, hence we can construct the following type derivation:

$$\textsc{Exp Fork}\ \dfrac{\Gamma; \Delta'_A \vdash D_1 : U_1 \qquad \dfrac{\Gamma; \Delta'_B \vdash D_2 : U_2 \qquad \Gamma; \Delta'_2 \vdash D_3 : V}{\Gamma; \Delta'_B, \Delta'_2 \vdash D_2 \upharpoonright D_3 : V}\ \textsc{Exp Fork}}{\Gamma; \Delta_A \vdash E \upharpoonright (E' \upharpoonright E'') : V}$$

since $\Gamma; \Delta_A, \Delta_1, \Delta_2, \Delta_3 \hookrightarrow \Gamma; \Delta'_A, (\Delta'_B, \Delta'_2)$ can be derived by Lemma A.8. Finally, we conclude $\Gamma; \Delta \vdash E \upharpoonright (E' \upharpoonright E'') : T$ 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 $D_1 \rightsquigarrow [\emptyset \mid D_1]$ and $D_2 \rightsquigarrow [\emptyset \mid D_2]$ by Lemma A.39, 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 $\mathsf{let}\ x = E'\ \mathsf{in}\ E'' \rightsquigarrow [\Delta_2 \mid \mathsf{let}\ x = D_2\ \mathsf{in}\ E'']$ by (Extr Let), hence we can construct the following type derivation:

$$\textsc{Exp Fork}\ \dfrac{\Gamma; \Delta'_A \vdash D_1 : U_1 \qquad \dfrac{\textsc{Exp Subsum}\ \dfrac{\Gamma; \Delta'_B \vdash D_2 : U_2 \qquad \Gamma; \Delta'_{12} \vdash U_2 <: U}{\Gamma; \Delta'_B, \Delta'_{12} \vdash D_2 : U} \qquad (\Gamma; \Delta'_2) \bullet x : U \vdash E'' : V}{\Gamma; \Delta'_B, \Delta'_{12}, \Delta'_2 \vdash \mathsf{let}\ x = D_2\ \mathsf{in}\ E'' : V}\ \textsc{Exp Let}}{\Gamma; \Delta_A \vdash E \Rsh (\mathsf{let}\ x = E'\ \mathsf{in}\ E'') : V}$$

since $\Gamma; \Delta_A, \Delta_1, \Delta_2 \hookrightarrow \Gamma; \Delta'_A, (\Delta'_B, \Delta'_{12}, \Delta'_2)$ can be derived by Lemma A.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:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $E \rightsquigarrow [\Delta_1 \mid D_1]$
- $\mathsf{let}\ x = E'\ \mathsf{in}\ E'' \rightsquigarrow [\Delta_2 \mid \mathsf{let}\ x = D_2\ \mathsf{in}\ E'']$ with $E' \rightsquigarrow [\Delta_2 \mid D_2]$
- $\Gamma; \Delta_A, \Delta_1, \Delta_2 \hookrightarrow \Gamma; \Delta'_1, \Delta'_2$
- $\Gamma; \Delta'_1 \vdash D_1 : U_1$
- $\Gamma; \Delta'_2 \vdash \mathsf{let}\ x = D_2\ \mathsf{in}\ E'' : V$

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 $E' \rightsquigarrow [\Delta_2 \mid D_2]$ implies $D_2 \rightsquigarrow [\emptyset \mid D_2]$ by Lemma A.39, 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 $E \Rsh E' \rightsquigarrow [\Delta_1, \Delta_2 \mid D_1 \Rsh D_2]$ by (Extr Fork). Moreover, we know that $E \rightsquigarrow [\Delta_1 \mid D_1]$ implies $D_1 \rightsquigarrow [\emptyset \mid D_1]$ by Lemma A.39, hence we can construct the following type derivation:

$$\text{Exp Let}\ \dfrac{\text{Exp Fork}\ \dfrac{\Gamma; \Delta'_1 \vdash D_1 : U_1 \qquad \Gamma; \Delta'_A \vdash D_2 : U_2}{\Gamma; \Delta'_1, \Delta'_A \vdash D_1 \Rsh D_2 : U_2} \qquad \dfrac{(1)}{(\Gamma; \Delta'_B, \Delta'_{22}) \bullet x : U_2 \vdash E'' : V}}{\Gamma; \Delta_A \vdash \mathsf{let}\ x = (E \Rsh E')\ \mathsf{in}\ E'' : V}$$

where $\Gamma; \Delta_A, \Delta_1, \Delta_2 \hookrightarrow \Gamma; (\Delta'_1, \Delta'_A), (\Delta'_B, \Delta'_{22})$ can be derived by Lemma A.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 $\Gamma; \Delta \vdash \mathsf{let}\ x = (E \Rsh E')\ \mathsf{in}\ E'' : T$ by (Exp Subsum).

□

The next simple lemma states that tautologies can be safely removed from any typing environment. This is used in some cases of the Subject Reduction proof, to deal with the logical formulas we explicitly introduce in the typing environment to make type-checking more precise (cf. Exp Split).

Lemma A.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 B.3. Moreover, the latter implies $\Gamma; \Delta \vdash \diamond$ again by Lemma B.3. Since $\Delta, F \vdash \Delta, F$ by Lemma A.2, we can derive $\Gamma; \Delta \hookrightarrow \Gamma; \Delta, F$ as follows:

$$\text{Rewrite}\ \dfrac{\Gamma; \Delta \vdash \diamond \qquad (*)\ \dfrac{\emptyset \vdash F \qquad \Delta, F \vdash \Delta, F}{\Delta \vdash \Delta, F} \qquad \Gamma; \Delta, F \vdash \diamond}{\Gamma; \Delta \hookrightarrow \Gamma; \Delta, F}$$

Here $(*)$ follows by using a standard Cut elimination argument. Since $\Gamma; \Delta, F \vdash E : T$, the conclusion $\Gamma; \Delta \vdash E : T$ follows by Lemma A.9. □

We can finally prove the Subject Reduction theorem. Its statement is remarkably simple: this is mainly due to our type system design, which discharges to the underlying affine logical framework all the complicated issues related to resource consumption. Thus, we do not need to explicitly track in the semantics which resources are consumed upon reduction, unlike to many other substructural type systems.

Theorem A.1 (Subject Reduction)**.** *Let* $fv(E) = \emptyset$*. If* $\Gamma; \Delta \vdash E : T$ *and* $E \rightarrow E'$*, then* $\Gamma; \Delta \vdash E' : T$.

Proof. By induction on the derivation of $E \rightarrow E'$. In the proof we implicitly appeal to Lemma B.3 and Lemma A.8 several times:

Case (Red Fun): assume $(\lambda x.\, E)\ N \rightarrow E\{N/x\}$ 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:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $\Gamma; \Delta_A \hookrightarrow \Gamma; \Delta_1, \Delta_2$
- $\Gamma; \Delta_1 \vdash \lambda x.\, E : x : T' \rightarrow U'$
- $\Gamma; \Delta_2 \vdash N : T'$

By Lemma A.31 we know that $\Gamma; \Delta_1 \vdash \lambda x.\, E : x : T' \rightarrow U'$ implies $(\Gamma; \Delta_1) \bullet x : T' \vdash E : U'$. Now notice that $x \notin dom(\Gamma)$ by Lemma B.3, hence $x \notin fv(\Delta_1)$ by Lemma A.10. By applying Lemma B.12, 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 $\mathsf{let}\ (x, y) = (M, N)\ \mathsf{in}\ E \rightarrow E\{M/x\}\{N/y\}$ 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:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $\Gamma; \Delta_A \hookrightarrow \Gamma; \Delta_1, \Delta_2$
- $\Gamma; \Delta_1 \vdash (M, N) : x : T' * U'$
- $(\Gamma; \Delta_2) \bullet x : T' \bullet y : U' \bullet !((x, y) = (M, N)) \vdash E : V$
- $\{x, y\} \cap fv(V) = \emptyset$

By Lemma A.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 $x \notin dom(\Gamma)$ by Lemma B.3, hence $x \notin fv(\Delta_2)$ by Lemma A.10. By applying Lemma B.12 twice and noting that $\{x, y\} \cap fv(V) = \emptyset$, 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 A.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 A.33 or Lemma A.34. and we get $\Gamma; \Delta_1 \vdash N : U'$. Now we notice that $\Gamma; \Delta_2 \vdash E' : V$ implies $fnfv(V) \subseteq dom(\Gamma)$ by Lemma B.3, hence the fact that $x \notin dom(\Gamma)$ implies $x \notin fv(V)$. Moreover, $x \notin dom(\Gamma)$ implies $x \notin fv(\Delta_2)$ by Lemma A.10. By applying Lemma B.12 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 A.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 $M = N \rightarrow \mathsf{false}$ 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 $a!M \Rsh a? \rightarrow M$ 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 B.3, hence $x \notin \mathit{fv}(\Delta_2)$ by Lemma A.10. By applying Lemma B.12 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 A.41 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 A.41 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 A.43. Since $D^* \Rrightarrow D''$ and $\Gamma; \Delta_1, \Delta'' \vdash D^* : U$, we get $\Gamma; \Delta_1, \Delta'' \vdash D'' : U$ by Lemma A.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 A.41 we know that $E \to E'$ and $E \rightsquigarrow^a [\Delta' \mid D]$ imply that there exist D', Δ'', D'', D^* such that $D \to D'$ and $E' \rightsquigarrow^a [\Delta', \Delta'' \mid D'']$ with $D' \rightsquigarrow^a [\Delta'' \mid D^*]$ and $D^* \Rrightarrow D''$. Since Lemma A.41 is depth-preserving, we can apply the inductive hypothesis and get $\Gamma, a \updownarrow U; \Delta_A, \Delta' \vdash D' : V$. Given that $D' \rightsquigarrow^a [\Delta'' \mid D^*]$ 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 A.43. By Lemma A.45 we get $\Gamma, a \updownarrow U; \Delta_A, \Delta', \Delta'' \vdash D'' : V$. Hence, we have:

- $E' \rightsquigarrow^a [\Delta', \Delta'' \mid D'']$
- $\Gamma, a \updownarrow U; \Delta_A, \Delta', \Delta'' \vdash D'' : V$

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 $E \Rsh E'' \to E' \Rsh E''$ with $E \to E'$ and $\Gamma; \Delta \vdash E \Rsh E'' : T$. The typing judgement must follow by an instance of (Exp Fork) after an instance of (Exp Subsum), hence it must be the case that $\Gamma; \Delta_A \vdash E \Rsh E'' : V$ and $\Gamma; \Delta_B \vdash V <: T$ with:

- $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_A, \Delta_B$
- $E \rightsquigarrow [\Delta' \mid D_1]$
- $E'' \rightsquigarrow [\Delta'' \mid D_2]$
- $\Gamma; \Delta_A, \Delta', \Delta'' \hookrightarrow \Gamma; \Delta_1, \Delta_2$
- $\Gamma; \Delta_1 \vdash D_1 : U$
- $\Gamma; \Delta_2 \vdash D_2 : V$

By Lemma A.41 we know that $E \to E'$ and $E \rightsquigarrow [\Delta' \mid D_1]$ imply that there exist D_1', Δ^*, D'', D^* such that $D_1 \to D_1'$ and $E' \rightsquigarrow [\Delta', \Delta^* \mid D'']$ with $D_1' \rightsquigarrow [\Delta^* \mid D^*]$ and $D^* \Rrightarrow D''$. Since Lemma A.41 is depth-preserving, we can apply the inductive hypothesis and get $\Gamma; \Delta_1 \vdash D_1' : U$. Given that $D_1' \rightsquigarrow [\Delta^* \mid D^*]$ and $\Gamma; \Delta_1 \vdash D_1' : U$, we get $\Gamma; \Delta_1, \Delta^* \vdash D^* : U$ by Lemma A.43. By Lemma A.45 we get $\Gamma; \Delta_1, \Delta^* \vdash D'' : U$. Hence, we have:

- $E' \rightsquigarrow [\Delta', \Delta^* \mid D'']$
- $E'' \rightsquigarrow [\Delta'' \mid D_2]$
- $\Gamma; \Delta_A, \Delta', \Delta^*, \Delta'' \hookrightarrow \Gamma; (\Delta_1, \Delta^*), \Delta_2$

- $\Gamma; \Delta_1, \Delta^* \vdash D'' : U$
- $\Gamma; \Delta_2 \vdash D_2 : V$

We can then apply rule (Exp Fork) to get $\Gamma; \Delta_A \vdash E' \Rsh E'' : V$. The conclusion $\Gamma; \Delta \vdash E' \Rsh E'' : T$ 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 A.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 A.45.

□

A.1.8 Proof of (robust) safety

We first show that well-typed structures are statically safe.

Lemma A.47 (Static Safety). *If* $\varepsilon; \emptyset \vdash \mathbf{S} : T$*, then* $\mathbf{S}$ *is statically safe.*

Proof. Consider an arbitrary structure:

$$(\nu a_1) \ldots (\nu a_r)((E_1 \Rsh E_2) \Rsh E_3) \Rsh E_4,$$

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 $F_1, \ldots, F_m \vdash F'_1 \otimes \ldots \otimes F'_n$.

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 A.38 we can simplify an arbitrary chain of extraction steps with decreasing index sets $\{a_1, \ldots, a_r\}, \ldots, \{a_r\}, \emptyset$ 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 A.39. By transitivity of subtyping and rewriting, using the previous facts, without loss of generality we have:

- $((E_1 \Rsh E_2) \Rsh E_3) \Rsh E_4 \rightsquigarrow^{\emptyset} [\Delta_1 \mid ((D_1 \Rsh E_2) \Rsh E_3) \Rsh E_4]$,
- where $E_1 \rightsquigarrow^{\emptyset} [\Delta_1 \mid D_1]$ 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 $\Gamma = a_1 \updownarrow T_1, \ldots, a_r \updownarrow T_r$
- $\Gamma; \Delta_{A_1} \vdash D_1 : U_1$ 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 $\mathbf{S}$. 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 A.8 we get $\Gamma; \Delta_1 \hookrightarrow \Gamma; \Delta'_{C_1}, \ldots, \Delta'_{C_n}$. By Lemma A.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 B.8. By inverting rule (Derive) this implies:

$$F_1, \ldots, F_m \vdash F'_1 \otimes \ldots \otimes F'_n.$$

□

The safety theorem below states that any well-typed expression is safe. Its proof is simple and relies on the previous results.

Restatement A.1 (of Theorem 2.2). *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 $\mathbf{S}$ such that $E \rightarrow^* E'$ and $E' \Rrightarrow \mathbf{S}$, it holds that $\mathbf{S}$ is statically safe.

By Theorem A.1, $\varepsilon; \emptyset \vdash E : T$ implies $\varepsilon; \emptyset \vdash E' : T$. By Lemma A.45, $E' \Rrightarrow \mathbf{S}$ implies $\varepsilon; \emptyset \vdash \mathbf{S} : T$. We can conclude that $\mathbf{S}$ is statically safe by Lemma A.47. □

The next lemma is important to show that any opponent is trivially well-typed: it identifies Un with a number of structural types built around Un itself.

Lemma A.48 (Universal Type). *If $\Gamma; \emptyset \vdash \diamond$, then $\Gamma; \emptyset \vdash T <:> \mathsf{Un}$ for all $T \in \{\mathsf{unit}, x : \mathsf{Un} \rightarrow \mathsf{Un}, x : \mathsf{Un} * \mathsf{Un}, \mathsf{Un} + \mathsf{Un}, \mu\alpha.\, \mathsf{Un}\}$.*

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 B.16. □

We can now show that any opponent is well-typed. The statement is slightly more general than expected, since we appeal to inductive reasoning in the proof.

Lemma A.49 (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 $O \rightsquigarrow^{\widetilde{a}} [\emptyset \mid O]$ 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 B.8.

- 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 A.48, using (Exp Subsum) whenever needed.

We conclude by an application of (Exp Subsum), using Lemma A.48. □

Finally, we can prove our main result of interest: if an expression E is assigned type Un by our type system, then it is robustly safe. The proof is an easy consequence of Theorem 2.2 and Lemma A.49.

Restatement A.2 (of Theorem 3.1)**.** *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 A.49 we know that $\Gamma; \emptyset \vdash O : \mathsf{Un}$. We can apply (Exp Subsum) and Lemma A.48 to derive:

$$\Gamma; \emptyset \vdash O : \mathsf{Un} \to \mathsf{Un}. \tag{A.1}$$

We can apply Lemma B.8 to $\varepsilon;\emptyset \vdash E : \mathsf{Un}$ and get $\Gamma;\emptyset \vdash E : \mathsf{Un}$. Assume now $E \rightsquigarrow [\Delta \mid D]$, by Lemma A.43 we have $\Gamma;\Delta \vdash D : \mathsf{Un}$. By Lemma B.8 we then get:

$$\Gamma, f : \mathsf{Un} \rightarrow \mathsf{Un}; \Delta \vdash D : \mathsf{Un}. \tag{A.2}$$

Since $O \rightsquigarrow^{\emptyset} [\emptyset \mid O]$, we can construct the following type derivation:

$$\dfrac{(\text{A.1})\ \dfrac{\cdots}{\Gamma;\emptyset \vdash O : \mathsf{Un} \rightarrow \mathsf{Un}} \qquad \dfrac{(\text{A.2})\ \dfrac{\cdots}{\Gamma, f : \mathsf{Un} \rightarrow \mathsf{Un}; \Delta \vdash D : \mathsf{Un}} \qquad \dfrac{\cdots}{\Gamma, f : \mathsf{Un} \rightarrow \mathsf{Un}, x : \mathsf{Un}; \emptyset \vdash f\ x : \mathsf{Un}}\ \textsc{Exp Appl}}{\Gamma, f : \mathsf{Un} \rightarrow \mathsf{Un}; \emptyset \vdash \mathsf{let}\ x = E\ \mathsf{in}\ f\ x : \mathsf{Un}}\ \textsc{Exp Let}}{\Gamma;\emptyset \vdash \mathsf{let}\ f = O\ \mathsf{in}\ \mathsf{let}\ x = E\ \mathsf{in}\ f\ x : \mathsf{Un}}\ \textsc{Exp Let}$$

Since $O\ E \rightsquigarrow^{\widetilde{b}} [\emptyset \mid O\ E]$ 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 2.2, we then know that $(\nu a_1)\ldots(\nu a_n)(O\ E)$ is safe. Since restrictions do not affect safety, we can conclude. □

A.2 Soundness of AF7$_{\text{alg}}$

In this section we prove the soundness (Theorem 2.4) and completeness (Theorem 2.5) of the algorithmic variant of our type system.

A.2.1 Logical properties

We begin by showing some important properties of the logic that play a pivotal role in the bottom-up construction of the unique proof obligation in the algorithmic type system and the corresponding proofs of soundness and completeness.

We use the following convenient notation to denote all logical entailment rules that modify the set of premises.

Definition A.2 (Left Rules $\vdash^{\mathsf{L}}$)**.** *We say $\Delta \vdash^{\mathsf{L}} F$ if the last applied logical entailment rule is a rule of the form (R-Left) or (Contr) or (Weak).*

Lemma A.50 (Implication)**.** *1. For all Δ, F, F' we have that $\Delta \vdash F \multimap F'$ iff $\Delta, F \vdash F'$.*

2. For all Γ, Δ, F, F' we have that $\Gamma;\Delta \vdash F \multimap F'$ iff $\Gamma;\Delta, F \vdash F'$.

Proof. 1. We show both directions separately:

- $\Delta \vdash F \multimap F' \Rightarrow \Delta, F \vdash F'$:

  We proceed by induction on the derivation of $\Delta \vdash F \multimap F'$. By inspection of the rules, we know that either $\Delta \vdash F \multimap F'$ by an application of ($\multimap$-Right) or (Ident) or (False) or $\Delta \vdash^{\mathsf{L}} F \multimap F'$.

Case ($\multimap$-Right): We know that $\Delta, F \vdash F'$ by the premise of the rule. We can immediately conclude.

Case (Ident): In this case $\Delta = F \multimap F'$. Using ($\multimap$-Left) and (Ident) we can immediately derive that

$$\frac{F \vdash F \qquad F' \vdash F'}{F \multimap F', F \vdash F'}.$$

Case (False): In this case we know $\Delta = \mathbf{0}$. We can apply (False) to derive that $\Delta \vdash F'$ and conclude by (Weak).

Case $\Delta \vdash^{\mathsf{L}} F \multimap F'$ for some rule R:
By part (A.2) we know

$$\frac{\Delta' \vdash F \multimap F'}{\Delta \vdash^{\mathsf{L}} F \multimap F'}\ R$$

for some environment Δ'. We apply the induction hypothesis and derive that $\Delta', F \vdash F'$.
We apply R to derive that $\Delta, F \vdash F'$, and conclude.

- $\Gamma; \Delta, F \vdash F' \Rightarrow \Gamma; \Delta \vdash F \multimap F'$:
In this case we can immediately conclude by an application of ($\multimap$-Right).

2. Follows immediately by the definition of (Derive) and (Form Env Entry), property (1) and Lemma B.3.

□

Lemma A.51 (Universal Quantification). *It holds that:*

1. *For all x, Δ, F such that $x \notin fv(\Delta)$, we have that $\Delta \vdash F$ iff $\Delta \vdash \forall x.F$.*

2. *For all Γ, x, T, Δ, F such that $\Gamma, x : \psi(T); \Delta \vdash \diamond$ and $x \notin fnfv(\Delta)$, we have that $\Gamma, x : \psi(T); \Delta \vdash F$ iff $\Gamma; \Delta \vdash \forall x.F$.*

Proof. 1. We assume that $x \notin fv(\Delta)$. We show both directions separately:

- $\Delta \vdash F \Rightarrow \Delta \vdash \forall x.F$:
Since $x \notin fv(\Delta)$ we can immediately apply ($\forall$-Right) and derive that

$$\Delta \vdash \forall x.F.$$

- $\Delta \vdash \forall x.F \Rightarrow \Delta \vdash F$:
We proceed by induction on the derivation of $\Delta \vdash \forall x.F$. By inspection of the rules, we know that either $\Delta \vdash \forall x.F$ by an application of ($\forall$-Right) or (Ident) or (False) or $\Delta \vdash^{\mathsf{L}} \forall x.F$.

Case ($\forall$-Right): By definition of the rule we know that $x \notin fv(\Delta)$ and $\Delta \vdash F$ and conclude.

Case (Ident:) In this case $\Delta = \forall x.F$. We know that $\forall x.F \vdash F$ by ($\forall$-Left) and (Ident).

Case (False): In this case we know $\Delta = \mathbf{0}$. We can apply (False), and derive $\Delta \vdash F$.

Case $\Delta \vdash^{\mathsf{L}} \forall x.F$ for some rule R:
By Definition A.2 we know that

$$\frac{\Delta' \vdash \forall x.F}{\Delta \vdash^{\mathsf{L}} \forall x.F}\, R\,.$$

for some Δ'. We apply the induction hypothesis and derive that $\Delta' \vdash F$.
We can apply R to derive that $\Delta \vdash F$.

2. Follows immediately by the definition of (Derive) and (Form Env Entry) and *fnfv*, property (1) and Lemma B.3.

□

A.2.2 Soundness and completeness of the algorithmic judgements

Lemma A.52 (Soundness and Completeness of Algorithmic Well-formedness). *For all* Γ*, the following holds true:*

1. $\Gamma \vdash_{\mathsf{alg}} \diamond$ *iff* $\Gamma; \emptyset \vdash \diamond$
2. *for all* T*,* $\Gamma \vdash_{\mathsf{alg}} T$ *iff* $\Gamma; \emptyset \vdash T$

Proof. By induction on the derivation of the respective well-formedness statement. □

Lemma A.53 (Soundness and Completeness of Algorithmic Kinding). *For all* Γ, T, k*, the following holds true:*

1. *for all* F, Δ *such that* $\Gamma \vdash_{\mathsf{alg}} T :: k; F$ *and* $\Gamma; \Delta \vdash F$*, we have that* $\Gamma; \Delta \vdash T :: k$*.*
2. *for all* Δ *such that* $\Gamma; \Delta \vdash T :: k$*, there exists* F *such that* $\Gamma \vdash_{\mathsf{alg}} T :: k; F$ *and* $\Gamma; \Delta \vdash F$*;*

Proof. 1. We first prove part (1). The proof proceeds by induction on the length of $\Gamma \vdash_{\mathsf{alg}} T :: k; F$. The base cases are (Kind Var Alg) and (Kind Unit Alg): they follow by an inspection of the typing rules and by Lemma A.52.

We now discuss the induction step.

Case (Kind Fun Alg): $\Gamma \vdash_{\mathsf{alg}} x : T \to U :: k; !F_1 \otimes !F_2$ is derived by $\Gamma \vdash_{\mathsf{alg}} T :: k; F_1$ and $\Gamma, x : \psi(T) \vdash_{\mathsf{alg}} U :: k; F_2$. We also know that $\Gamma; \Delta \vdash !F_1 \otimes !F_2$.

To show: $\Gamma; \Delta \vdash x : T \to U :: k$ We know that that $\Gamma; \Delta \hookrightarrow \Gamma; !F_1, !F_2$ by (Rewrite) and (Derive) and $\Gamma; !F_1 \vdash F_1$, and $\Gamma; !F_2 \vdash F_2$ by (Ident), (!-Left), and (Derive). By induction hypothesis, $\Gamma; !F_1 \vdash T :: \overline{k}$ and $\Gamma, x : \psi(T); !F_2 \vdash U :: k$. The result follows from (Kind Fun).

Case (Kind Pair Alg): The proof is analogous to the one for (Kind Fun Alg).

Case (Kind Sum Alg): The proof is analogous to the one for (Kind Fun Alg).

Case (Kind Rec Alg): $\Gamma \vdash_{\mathsf{alg}} \mu\alpha. T :: k; !F$ is proved by $\Gamma, \alpha :: k \vdash_{\mathsf{alg}} T :: k; F$. We also know that $\Gamma; \Delta \vdash !F$.

To show: $\Gamma; \Delta \vdash \mu\alpha. T :: k$ By (Rewrite) and (Derive) and Lemma B.3, $\Gamma; \Delta \hookrightarrow \Gamma; !F$ and $\Gamma; !F \vdash F$ by (Ident), (!-Left), and (Derive). By induction hypothesis, $\Gamma, \alpha :: k; !F \vdash T :: k$. The result follows from (Kind Rec).

Case (Kind Refine Public Alg): The proof follows immediately from the induction hypothesis.

Case (Kind Refine Tainted Alg): $\Gamma \vdash_{\mathsf{alg}} T :: \mathsf{tnt}; (\forall x.\mathit{forms}(x : T)) \otimes F'$, where T is refined is proved by $\Gamma \vdash_{\mathsf{alg}} \psi(T) :: \mathsf{tnt}; F'$, $\Gamma, x : \psi(T) \vdash_{\mathsf{alg}} \diamond$, and $\Gamma \vdash_{\mathsf{alg}} T$. We also know that $\Gamma; \Delta \vdash (\forall x.\mathit{forms}(x : T)) \otimes F'$.

To show: $\Gamma; \Delta \vdash T :: \mathsf{tnt}$ By (Rewrite), (Derive), and Lemma B.3 we know that that $\Gamma; \Delta \hookrightarrow \Gamma; (\forall x.\mathit{forms}(x : T)), F'$ and $\Gamma; (\forall x.\mathit{forms}(x : T)) \vdash (\forall x.\mathit{forms}(x : T))$ and $\Gamma; F' \vdash F'$ by (Ident) and (Derive). By induction hypothesis, $\Gamma; F' \vdash_{\mathsf{alg}} \psi(T) :: \mathsf{tnt}$. By ($\forall$-Left), (Ident), and (Derive), $\Gamma, x : \psi(T); (\forall x.\mathit{forms}(x : T)) \vdash \mathit{forms}(x : T)$. The result follows from (Kind Refine Tainted).

2. We now prove part (2). The proof proceeds by induction on the length of $\Gamma; \Delta \vdash T :: k$. The base cases are (Kind Var) and (Kind Unit): they follow by an inspection of the typing rules, by Lemma A.52, and by observing that $\Gamma; \Delta \vdash \mathbf{1}$ for any Δ such that $\Gamma; \Delta \vdash \diamond$. We now discuss the induction step.

Case (Kind Fun): $\Gamma; \Delta \vdash x : T \to U :: k$ is proved by $\Gamma; !\Delta_1 \vdash T :: \overline{k}$, $\Gamma, x : \psi(T); !\Delta_2 \vdash U :: k$, and $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2$. By induction hypothesis, there exist F_1, F_2 such that $\Gamma \vdash_{\mathsf{alg}} T :: \overline{k}; F_1$, $\Gamma, x : \psi(T) \vdash_{\mathsf{alg}} U :: k; F_2$, $\Gamma; !\Delta_1 \vdash F_1$, and $\Gamma; !\Delta_2 \vdash F_2$.

To show: $\Gamma \vdash_{\mathsf{alg}} x : T \to U :: k; !F_1 \otimes !F_2$ The result follows from (Kind Fun Alg).

To show: $\Gamma; \Delta \vdash !F_1 \otimes !F_2$ We use (Derive) as needed. By (!-Right) we can derive that $\Gamma; !\Delta_1 \vdash !F_1$ and $\Gamma; !\Delta_2 \vdash !F_2$. By ($\otimes$-Right), $\Gamma; !\Delta_1, !\Delta_2 \vdash !F_1 \otimes !F_2$. The result follows from Lemma A.9.

Case (Kind Pair): The proof is analogous to the one for (Kind Fun).

Case (Kind Sum): The proof is analogous to the one for (Kind Fun).

Case (Kind Rec): $\Gamma; \Delta \vdash \mu\alpha.T :: k$ is proved by $\Gamma, \alpha :: k; !\Delta' \vdash T :: k$ and $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$. By induction hypothesis, there exists F such that $\Gamma, \alpha :: k \vdash_{\mathsf{alg}} T :: k; F$ and $\Gamma; !\Delta' \vdash F$.

To show: $\Gamma \vdash_{\mathsf{alg}} \mu\alpha.T :: k; !F$ The result follows from (Kind Rec Alg).

To show: $\Gamma; \Delta \vdash !F$ Using (Derive) and (!-Right) we can derive that $\Gamma; !\Delta' \vdash !F$. The result follows from Lemma A.9.

Case (Kind Refine Public): The proof follows immediately from the induction hypothesis.

Case (Kind Refine Tainted): $\Gamma; \Delta \vdash T :: \mathsf{tnt}$, where T is refined is proved by $\Gamma; \Delta_1 \vdash \psi(T) :: \mathsf{tnt}$, $\Gamma, x : \psi(T); \Delta_2 \vdash \mathit{forms}(x : T)$, and $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$. By induction hypothesis, there exists F' such that $\Gamma \vdash_{\mathsf{alg}} \psi(T) :: \mathsf{tnt}; F'$ and $\Gamma; \Delta_1 \vdash F'$.

To show: $\Gamma \vdash_{\mathsf{alg}} T :: \mathsf{tnt}; (\forall x.\mathit{forms}(x : T)) \otimes F'$ The result follows from (Kind Refine Tainted Alg) and Lemma B.3.

To show: $\Gamma; \Delta \vdash (\forall x.\mathit{forms}(x : T)) \otimes F'$ By Lemma B.3, $\Gamma, x : \psi(T); \Delta_2 \vdash \diamond$ and, thus, $x \notin \mathit{dom}(\Gamma)$. Since $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$, we also have that $\Gamma; \Delta_1, \Delta_2 \vdash \diamond$ and, thus, $x \notin \mathit{fnfv}(\Delta_2) \subseteq \mathit{dom}(\Gamma)$. By Lemma A.51, $\Gamma; \Delta_2 \vdash \forall x.\mathit{forms}(x : T)$. By ($\otimes$-Right), $\Gamma; \Delta_1, \Delta_2 \vdash (\forall x.\mathit{forms}(x : T)) \otimes F'$. The result follows from Lemma A.9.

□

In order to prove soundness and completeness of subtyping we will proceed in two steps: we first introduce an intermediate algorithmic variant of the standard subtyping relation $\Gamma; \Delta \vdash \overline{T} <:_{\mathsf{alg}} \overline{U}$ for annotated types that extends the standard one by adding the side conditions in (Sub Refl) and (Sub Pub Tnt) present in algorithmic subtyping and provides three disjoint rules for subtyping two iso-recursive types, following the insights given in Section 2.10.5.

We will then show that we can find annotations to prove the standard subtyping and intermediate subtyping equivalent and show the soundness and completeness of algorithmic subtyping with respect to intermediate subtyping $<:_{\mathsf{alg}}$.

The full definition of the intermediate subtyping rules can be found in Table A.2. The rules make use of the previously introduced annotated types $\overline{T}$ that might contain type annotation SPT. As in algorithmic subtyping, we assume the function ψ to extend to annotated types and we write $T = \langle\overline{T}\rangle$ to denote the type that results from erasing all type annotations from $\overline{T}$. We say T and $\overline{T}$ are equal up to type annotations.

The soundness and completeness proof of intermediate subtyping makes use of the following propositions and lemmas. We write $\overline{T} = \{x_m : \ldots \{x_2 : \{x_1 : \overline{U} \mid F_1\} \mid F_2\} \ldots \mid F_m\}$ to denote nested (annotated) refinement types, for $m = 0$ this notation simply denotes the annotated type $\overline{U}$.

The following proposition states that all types are also annotated types by construction, which we will use implicitly throughout the following proofs.

Proposition A.1 (Types and Annotated Types)**.** *Let T be a type. Then T is also an annotated type, such that $\langle T\rangle = T$.*

Lemma A.54 (Refinement Erasure of Annotated Types)**.** *For all types T and annotated types $\overline{T}$ such that $T = \langle\overline{T}\rangle$ it holds that $\psi(T) = \psi(\langle\overline{T}\rangle) = \langle\psi(\overline{T})\rangle$.*

Proof. Proof by induction on the structure of $\overline{T}$ using the definitions of ψ and the erasure of annotations $\langle\bullet\rangle$. □

Lemma A.55 (Nested Refinements)**.** *For all types T it holds that there exist $m \geq 0$ and T_{min} and $F_1, \ldots, F_m, x_1, \ldots, x_m$ (if $m > 0$) such that $T = \{x_m : \ldots\{x_2 : \{x_1 : T_{min} \mid F_1\} \mid F_2\} \ldots \mid F_m\}$ and $\psi(T) = \psi(T_{min}) = T_{min}$.*

Proof. Proof by induction on the structure of T using the definition of ψ. □

Lemma A.56 (Annotated Refinement Types)**.** *For all types $T = \{x_m : \ldots\{x_2 : \{x_1 : T_{min} \mid F_1\} \mid F_2\} \ldots \mid F_m\}$, where $\psi(T) = \psi(T_{min}) = T_{min}$, and all annotated types $\overline{T_{min}}$ and $\overline{T} = \{x_m : \ldots\{x_2 : \{x_1 : \overline{T_{min}} \mid F_1\} \mid F_2\} \ldots \mid F_m\}$ such that $T_{min} = \langle\overline{T_{min}}\rangle$ it holds that*

- *$\langle\overline{T}\rangle = T$ and*
- *$\psi(\overline{T}) = \overline{T_{min}}$.*

Proof. The first statement follows by induction on m using the definition of $\langle\bullet\rangle$. For the second statement we first note that $T_{min} = \psi(T_{min})$. By the definition of $\langle\bullet\rangle$ we know that T_{min} and $\overline{T_{min}}$ are equal up to typing annotations and thus by definition of ψ it must be the case that $\overline{T_{min}} = \psi(\overline{T_{min}})$. We conclude by induction on m, using the definition of ψ. □

Lemma A.57 (Soundness and Completeness of Intermediate Subtyping)**.** *For all Γ, Δ, T, U it holds that:*

1. *If $\Gamma; \Delta \vdash T$ then $\Gamma; \Delta \vdash T <:_{\mathsf{alg}} T$.*
2. *If there exist Δ_1, Δ_2 such that $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$ and $\Gamma; \Delta_1 \vdash T :: \mathsf{pub}$ and $\Gamma; \Delta_2 \vdash U :: \mathsf{tnt}$ then there exist annotated types $\overline{T}$ and $\overline{U}$ such that $T = \langle\overline{T}\rangle$, $U = \langle\overline{U}\rangle$, and $\Gamma; \Delta \vdash \overline{T} <:_{\mathsf{alg}} \overline{U}$.*
3. *If $\Gamma; \Delta \vdash \overline{T} <:_{\mathsf{alg}} \overline{U}$ and $T = \langle\overline{T}\rangle$, $U = \langle\overline{U}\rangle$, then $\Gamma; \Delta \vdash T <: U$.*
4. *If $\Gamma; \Delta \vdash T <: U$ then there exist $\overline{T}, \overline{U}$ such that $T = \langle\overline{T}\rangle$, $U = \langle\overline{U}\rangle$, and $\Gamma; \Delta \vdash \overline{T} <:_{\mathsf{alg}} \overline{U}$.*

Proof. Throughout the proof we make us of Proposition A.1 that allows us to consider each type T as an annotated type such that $\langle T\rangle = T$.

1. Proof by induction on the structure of T.

   If $T \in \{\alpha, \mathsf{unit}\}$ we can immediately conclude by an application of (Sub Refl *).

If T is an iso-recursive type we can immediately conclude by an application of (SUB REFL REC *).

If T is a pair, sum, or function type the proofs follow the same strategy, which we show exemplarily for $T = x : T_1 * T_2$. We know that $\Gamma; \Delta \hookrightarrow \Gamma; \emptyset, \emptyset$ by Lemma A.8. Using Lemma B.3 we apply the induction hypothesis to T_1 to derive that $\Gamma; \emptyset \vdash T_1 <:_{\mathsf{alg}} T_1$ and to T_2 to derive that $\Gamma, x : \psi(T_1); \emptyset \vdash T_2 <:_{\mathsf{alg}} T_2$. We conclude by an application of (SUB PAIR *).

If $T = \{x : U \mid F\}$ we can apply the induction hypothesis and Lemma B.3 to derive that $\Gamma; \emptyset \vdash \psi(T) <:_{\mathsf{alg}} \psi(T)$. It is easy to see that $\Gamma, y : \psi(T); \mathit{forms}(y : T) \vdash \mathit{forms}(y : T)$. Since we know that $\Gamma; \Delta \hookrightarrow \Gamma; \emptyset$ by Lemma A.8 we can conclude by an application of (SUB REFINE *).

2. Proof by induction on the structure of T. Note that whenever it is the case that $T \neq_{\top} U$ no annotations are necessary and we can immediately conclude by an application of (SUB PUB TNT *). In the following we thus assume that T and U share the same top-level constructor, or that one of them is refined.

Case In the case where $T = U$ and $T \in \{\alpha, \mathsf{unit}\}$ we can immediately conclude by applying (SUB REFL *) to $\overline{T} := T$ and $\overline{U} := U$ using Lemma B.3.

Case In the case where T is of the form $\mu\alpha.\,T_1$ and U is of the form $\mu\alpha.\,U_1$ for some T_1, U_1 we can immediately conclude by annotating U with SPT such that we can apply (SUB PUB TNT REC *) to $\overline{T} := T$ and $\overline{U} := U_{\mathsf{SPT}}$. We note that $\langle U_{\mathsf{SPT}} \rangle = U$ by definition.

Case The proofs of the cases where T, U are a couple of pair, sum, or function types all follow the same strategy, which we show exemplarily for the case $T = x : T_1 * T_2$ and $U = x : U_1 * U_2$ (note that the reasoning for sum types will be somewhat simplified).

We know that $\Gamma; \Delta_1 \vdash x : T_1 * T_2 :: \mathsf{pub}$ which by definition of the only applicable kinding rule (KIND PAIR) implies that there must exist Δ_{11}, Δ_{12} such that $\Gamma; \Delta_1 \hookrightarrow \Gamma; !\Delta_{11}, !\Delta_{12}$ and $\Gamma; !\Delta_{11} \vdash T_1 :: \mathsf{pub}$ and $\Gamma, x : \psi(T_1); !\Delta_{12} \vdash T_2 :: \mathsf{pub}$.

Using the same reasoning, we know that $\Gamma; \Delta_2 \vdash x : U_1 * U_2 :: \mathsf{tnt}$ by definition of the only applicable kinding rule (KIND PAIR) implies that there must exist Δ_{21}, Δ_{22} such that $\Gamma; \Delta_2 \hookrightarrow \Gamma; !\Delta_{21}, !\Delta_{22}$ and $\Gamma; !\Delta_{21} \vdash U_1 :: \mathsf{tnt}$ and $\Gamma, x : \psi(U_1); !\Delta_{22} \vdash U_2 :: \mathsf{tnt}$.

Applying the induction hypothesis to $\Gamma; !\Delta_{11} \vdash T_1 :: \mathsf{pub}$ and $\Gamma; !\Delta_{21} \vdash U_1 :: \mathsf{tnt}$ (implicitly using Lemma A.8) lets us derive that there exist annotated types $\overline{T_1}, \overline{U_1}$ such that

$$\Gamma; !\Delta_{11}, !\Delta_{21} \vdash \overline{T_1} <:_{\mathsf{alg}} \overline{U_1},$$

where $T_1 = \langle \overline{T_1} \rangle$ and $U_1 = \langle \overline{U_1} \rangle$.

By Lemma A.16 we know that $\Gamma, x : \psi(U_1); !\Delta_{22} \vdash U_2 :: \mathsf{tnt}$ implies $\Gamma, x : \psi(T_1); !\Delta_{22} \vdash U_2 :: \mathsf{tnt}$. We apply the induction hypothesis to the latter statement and $\Gamma, x : \psi(T_1); !\Delta_{12} \vdash T_2 :: \mathsf{pub}$, allowing us to derive that there exist annotated types $\overline{T_2}, \overline{U_2}$ such that

$$\Gamma, x : \psi(T_1); !\Delta_{12}, !\Delta_{22} \vdash \overline{T_2} <:_{\mathsf{alg}} \overline{U_2},$$

where $T_2 = \langle\overline{T_2}\rangle$ and $U_2 = \langle\overline{U_2}\rangle$.

By Lemma A.8 we know that

$$\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_{11}, !\Delta_{21}, !\Delta_{12}, !\Delta_{22}$$

and can thus conclude by an application of (SUB PAIR *), for $\overline{T} := x : \overline{T_1} * \overline{T_2}$ and $\overline{U} := x : \overline{U_1} * \overline{U_2}$. By definition of $\langle\bullet\rangle$ we know that $\langle\overline{T}\rangle = x : \langle\overline{T_1}\rangle * \langle\overline{T_2}\rangle = x : T_1 * T_2 = T$ and analogously $\langle\overline{U}\rangle = U$.

Case In the case that T or U (or both) are refined we proceed as follows. First we note that by Lemma A.55 there must thus exist $m \geq 0, n \geq 0$ (where $m > 0$ or $n > 0$ or both) and T_{min}, U_{min} and $F_1, \ldots, F_m, x_1, \ldots, x_m$ and $F'_1, \ldots, F'_n, x'_1, \ldots, x'_n$ such that $T = \{x_m : \ldots \{x_2 : \{x_1 : T_{min} \mid F_1\} \mid F_2\} \ldots \mid F_m\}$ and $U = \{x'_n : \ldots \{x'_2 : \{x'_1 : U_{min} \mid F'_1\} \mid F'_2\} \ldots \mid F'_n\}$ and $\psi(T) = \psi(T_{min}) = T_{min}$ and $\psi(U) = \psi(U_{min}) = U_{min}$.

We know that $\Gamma; \Delta_1 \vdash T :: \mathsf{pub}$, which by Lemma A.13 implies that there exist Δ_{11}, Δ_{12} such that $\Gamma; \Delta_1 \hookrightarrow \Gamma; !\Delta_{11}, \Delta_{12}$ and $\Gamma; !\Delta_{11} \vdash \psi(T) :: \mathsf{pub}$.

Furthermore, by Lemma A.13 we know that $\Gamma; \Delta_2 \vdash U :: \mathsf{tnt}$ implies that there exist Δ_{21}, Δ_{22} such that $\Gamma; \Delta_2 \hookrightarrow \Gamma; !\Delta_{21}, \Delta_{22}$ and $\Gamma; !\Delta_{21} \vdash \psi(U) :: \mathsf{tnt}$ and $\Delta_{22} \vdash \mathit{forms}(y : U)$ for some $y \notin \mathit{fv}(\Gamma)$.

We can apply the induction hypothesis to $\Gamma; !\Delta_{11} \vdash \psi(T) :: \mathsf{pub}$ (which is equivalent to $\Gamma; !\Delta_{11} \vdash T_{min} :: \mathsf{pub}$) and $\Gamma; !\Delta_{21} \vdash \psi(U) :: \mathsf{tnt}$ (which is equivalent to $\Gamma; !\Delta_{21} \vdash U_{min} :: \mathsf{tnt}$) to derive that there exist $\overline{T_{\mathsf{min}}}, \overline{U_{\mathsf{min}}}$ such that

$$\Gamma; !\Delta_{11}, !\Delta_{21} \vdash \overline{T_{\mathsf{min}}} <:_{\mathsf{alg}} \overline{U_{\mathsf{min}}},$$

where $\psi(T) = T_{\mathsf{min}} = \langle\overline{T_{\mathsf{min}}}\rangle$ and $\psi(U) = U_{\mathsf{min}} = \langle\overline{U_{\mathsf{min}}}\rangle$.

We construct the annotated types $\overline{T}$ and $\overline{U}$ as follows: $\overline{T} := \{x_m : \ldots \{x_2 : \{x_1 : \overline{T_{min}} \mid F_1\} \mid F_2\} \ldots \mid F_m\}$ and $\overline{U} := \{x'_n : \ldots \{x'_2 : \{x'_1 : \overline{U_{min}} \mid F'_1\} \mid F'_2\} \ldots \mid F'_n\}$. By Lemma A.56 we know that $T = \langle\overline{T}\rangle$ and $U = \langle\overline{U}\rangle$ and $\psi(\overline{T}) = \psi(\overline{T_{min}}) = \overline{T_{min}}$ and $\psi(\overline{U}) = \psi(\overline{U_{min}}) = \overline{U_{min}}$.

Weakening in the logic allows us to derive that $\Delta_{22} \vdash \mathit{forms}(y : U)$ implies that $\Delta_{22}, \mathit{forms}(y : T) \vdash \mathit{forms}(y : U)$. By applying (DERIVE) and Lemma B.3 we can conclude that

$$\Gamma, y : \psi(T); \Delta_{22}, \mathit{forms}(y : T) \vdash \mathit{forms}(y : U).$$

We use Lemma A.8 to derive that

$$\Gamma; \Delta \hookrightarrow \Gamma; (!\Delta_{11}, !\Delta_{21}), \Delta_{22}$$

and conclude that $\Gamma; \Delta \vdash \overline{T} <:_{\mathsf{alg}} \overline{U}$ by an application of (SUB REFINE *).

3. Straightforward induction on the derivation of $\Gamma; \Delta \vdash \overline{T} <:_{\mathsf{alg}} \overline{U}$.

   In the case where the last applied rule was Sub Refl * we know that $\overline{T} = \overline{U} = T = U$ and $T \in \{\alpha, \mathsf{unit}\}$ and $\Gamma; \Delta \vdash T$, which allows us to conclude by an application of Sub Refl.

   In the case where the last applied rule was (Sub Pub Tnt *) we know that $T = \overline{T}$ and $U = \overline{U}$ and $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$ for some Δ_1, Δ_2 such that $\Gamma; \Delta_1 \vdash T :: \mathsf{pub}$ and $\Gamma; \Delta_2 \vdash U :: \mathsf{tnt}$. We can immediately conclude by an application of (Sub Pub Tnt).

   In the case where the last applied rule (Sub Refl Rec *) we match the necessary premise to conclude with an application of (Sub Refl), in the case of (Sub Pub Tnt Rec *) we can conclude with an application of (Sub Pub Tnt).

   In all other cases R * we apply the induction hypothesis to the premises of R * and conclude by an application of R. Note that for (Sub Refine *) we make use of Lemma A.54, which lets us deduce that $\psi(T) = \langle \psi(\overline{T}) \rangle$ and $\psi(U) = \langle \psi(\overline{U}) \rangle$.

4. Proof by induction on the derivation of $\Gamma; \Delta \vdash T <: U$. We distinguish upon the last applied subtyping rule R:

Case (Sub Refl): In this case we can immediately conclude by an application of statement (1) of this lemma, choosing $\overline{T} := T$ and $\overline{U} := U$.

Case (Sub Pub Tnt): In this case we can immediately conclude by an application of statement (2) of this lemma.

Case (Sub Pos Rec) and $T = U$: In this case we can immediately conclude by an application of (Sub Refl Rec *), choosing $\overline{T} := T$ and $\overline{U} := U$.

Case (Sub Pos Rec) and $T \neq U$: In this case we know that $T = \mu\alpha.\, T_1$ and $U = \mu\alpha.\, U_1$. We apply the induction hypothesis to the subtyping premise of the rule to derive that $\Gamma, \alpha; !\Delta' \vdash \overline{T_1} <:_{\mathsf{alg}} \overline{U_1}$ for some $\overline{T_1}, \overline{U_1}$ such that $T_1 = \langle \overline{T_1} \rangle$ and $U_1 = \langle \overline{U_1} \rangle$ and $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$. We conclude by an application of (Sub Pos Rec *), choosing $\overline{T} := \mu\alpha.\, \overline{T_1}$ and $\overline{U} := \mu\alpha.\, \overline{U_1}$. Note that by definition of $\langle \bullet \rangle$ we know that $\langle \overline{T} \rangle = \mu\alpha.\, \langle \overline{T_1} \rangle = \mu\alpha.\, T_1 = T$ and analogously $\langle \overline{U} \rangle = U$.

Case R is (Sub Pair), (Sub Fun), (Sub Sum): These cases follow similarly to the previous case of distinct iso-recursive types by applying the induction hypothesis to the subtyping premises of the rule and concluding by an application of R *.

Case (Sub Refine): In this case we know that T or U (or both) are refined. First we note that by Lemma A.55 there exist $m \geq 0, n \geq 0$ (where $m > 0$ or $n > 0$ or both) and T_{min}, U_{min} and $F_1, \ldots, F_m, x_1, \ldots, x_m$ and $F'_1, \ldots, F'_n, x'_1, \ldots, x'_n$ such that $T = \{x_m : \ldots \{x_2 : \{x_1 : T_{min} \mid F_1\} \mid F_2\} \ldots \mid F_m\}$ and $U = \{x'_n : \ldots \{x'_2 : \{x'_1 : U_{min} \mid F'_1\} \mid F'_2\} \ldots \mid F'_n\}$ and $\psi(T) = \psi(T_{min}) = T_{min}$ and $\psi(U) = \psi(U_{min}) = U_{min}$.

By the definition of (Sub Refine) we know that there exist Δ_1, Δ_2 such that $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, \Delta_2$ and $\Gamma; \Delta_1 \vdash \psi(T) <: \psi(U)$ and $\Gamma, x : \psi(T); \Delta_2, \mathit{forms}(y : T) \vdash \mathit{forms}(y : U)$.

We can apply the induction hypothesis to $\Gamma; \Delta_1 \vdash \psi(T) <: \psi(U)$ (which is equivalent to $\Gamma; \Delta_1 \vdash T_{min} <: U_{min}$) to derive that there exist $\overline{T_{\mathsf{min}}}, \overline{U_{\mathsf{min}}}$ such that

$$\Gamma; \Delta_1 \vdash \overline{T_{\mathsf{min}}} <:_{\mathsf{alg}} \overline{U_{\mathsf{min}}},$$

where $\psi(T) = T_{\mathsf{min}} = \langle \overline{T_{\mathsf{min}}} \rangle$ and $\psi(U) = U_{\mathsf{min}} = \langle \overline{U_{\mathsf{min}}} \rangle$.

We construct the annotated types $\overline{T}$ and $\overline{U}$ as follows: $\overline{T} := \{x_m : \ldots \{x_2 : \{x_1 : \overline{T_{min}} \mid F_1\} \mid F_2\} \ldots \mid F_m\}$ and $\overline{U} := \{x'_n : \ldots \{x'_2 : \{x'_1 : \overline{U_{min}} \mid F'_1\} \mid F'_2\} \ldots \mid F'_n\}$. By Lemma A.56 we know that $T = \langle \overline{T} \rangle$ and $U = \langle \overline{U} \rangle$ and $\psi(\overline{T}) = \psi(\overline{T_{min}}) = \overline{T_{min}}$ and $\psi(\overline{U}) = \psi(\overline{U_{min}}) = \overline{U_{min}}$.

We conclude that $\Gamma; \Delta \vdash \overline{T} <:_{\mathsf{alg}} \overline{U}$ by an application of (Sub Refine *).

□

Lemma A.58 (Soundness and Completeness of Algorithmic Subtyping). *1. For all $\Gamma, \Delta, \overline{T}, \overline{U}$, the following holds true:*

(a) For all F such that $\Gamma \vdash_{\mathsf{alg}} \overline{T} <: \overline{U}; F$ and $\Gamma; \Delta \vdash F$, we have that $\Gamma; \Delta \vdash \overline{T} <:_{\mathsf{alg}} \overline{U}$.

(b) If $\Gamma; \Delta \vdash \overline{T} <:_{\mathsf{alg}} \overline{U}$, then there exists F such that $\Gamma \vdash_{\mathsf{alg}} \overline{T} <: \overline{U}; F$ and $\Gamma; \Delta \vdash F$;

2. For all Γ, Δ, T, U, the following holds true:

(a) For all $\overline{T}, \overline{U}, F$ such that $\Gamma \vdash_{\mathsf{alg}} \overline{T} <: \overline{U}; F$ and $\Gamma; \Delta \vdash F$ and $T = \langle \overline{T} \rangle$, $U = \langle \overline{U} \rangle$, we have that $\Gamma; \Delta \vdash T <: U$.

(b) If $\Gamma; \Delta \vdash T <: U$ then there exist $\overline{T}, \overline{U}, F$ such that $T = \langle \overline{T} \rangle$, $U = \langle \overline{U} \rangle$, and $\Gamma \vdash_{\mathsf{alg}} \overline{T} <: \overline{U}; F$ and $\Gamma; \Delta \vdash F$.

Proof. 1. We first prove part (1a). The proof proceeds by induction on the length of $\Gamma \vdash_{\mathsf{alg}} \overline{T} <: \overline{U}; F$. We use the non-annotated types $T := \langle \overline{T} \rangle$ and $U := \langle \overline{U} \rangle$ whenever applicable.

The base cases (Sub Refl Alg) and (Sub Refl Rec Alg) follow by an inspection of the typing rules and by Lemma A.52.

We now discuss the induction step by analyzing the last applied algorithmic subtyping rule.

Case (Sub Pub Tnt Alg): $\Gamma \vdash_{\mathsf{alg}} T <: U; F_1 \otimes F_2$ is proved by $\Gamma \vdash_{\mathsf{alg}} T :: \mathsf{pub}; F_1$ and $\Gamma \vdash_{\mathsf{alg}} U :: \mathsf{tnt}; F_2$, where $T = \overline{T}$ and $U = \overline{U}$ are not annotated. We also know that $\Gamma; \Delta \vdash F_1 \otimes F_2$ and $T \neq_{\top} U$.

To show: $\Gamma; \Delta \vdash T <:_{\mathsf{alg}} U$ By (Rewrite), (Derive), and Lemma B.3 we know that $\Gamma; \Delta \hookrightarrow \Gamma; F_1, F_2$ and $\Gamma; F_1 \vdash F_1$ and $\Gamma; F_2 \vdash F_2$ by (Ident) and (Derive). By Lemma A.53, $\Gamma; F_1 \vdash T :: \mathsf{pub}$ and $\Gamma; F_2 \vdash U :: \mathsf{tnt}$. The result follows from (Sub Pub Tnt *).

Case (Sub Pub Tnt Rec Alg): The proof follows the same structure as the one for (Sub Pub Tnt Alg) and concludes with an application of (Sub Pub Tnt Rec *).

Case (Sub Fun Alg): $\Gamma \vdash_{\mathsf{alg}} x : \overline{T_1} \to \overline{T_2} <: x : \overline{U_1} \to \overline{U_2}; !F_1 \otimes !F_2$ is derived by $\Gamma \vdash_{\mathsf{alg}} \overline{U_1} <: \overline{T_1}; F_1$ and $\Gamma, x : \psi(U_1) \vdash_{\mathsf{alg}} \overline{T_2} <: \overline{U_2}; F_2$. We also know that $\Gamma; \Delta \vdash !F_1 \otimes !F_2$.

To show: $\Gamma; \Delta \vdash x : \overline{T_1} \to \overline{T_2} <:_{\mathsf{alg}} x : \overline{U_1} \to \overline{U_2}$ By (Rewrite), (Derive), and Lemma B.3 we know that $\Gamma; \Delta \hookrightarrow \Gamma; !F_1, !F_2$ and $\Gamma; !F_1 \vdash F_1$, and $\Gamma; !F_2 \vdash F_2$ by (Ident), (!-Left), and (Derive).
By induction hypothesis, $\Gamma; !F_1 \vdash \overline{U_1} <:_{\mathsf{alg}} \overline{T_1}$ and $\Gamma, x : \psi(U_1); !F_2 \vdash \overline{T_2} <:_{\mathsf{alg}} \overline{U_2}$. The result follows from (Sub Fun *).

Case (Sub Pair Alg): The proof is analogous to the one for (Sub Fun Alg).

Case (Sub Sum Alg): The proof is analogous to the one for (Sub Fun Alg).

Case (Sub Pos Rec Alg): $\Gamma \vdash_{\mathsf{alg}} \mu\alpha.\overline{T_1} <: \mu\alpha.\overline{U_1}; !F$ is proved by $\Gamma, \alpha \vdash_{\mathsf{alg}} \overline{T_1} <: \overline{U_1}; F$. We also know that $\Gamma; \Delta \vdash !F$ and that α occurs only positively in $\overline{T_1}$ and $\overline{U_1}$.

To show: $\Gamma; \Delta \vdash \mu\alpha.\overline{T_1} <:_{\mathsf{alg}} \mu\alpha.\overline{U_1}$ By (Rewrite), (Derive), and Lemma B.3 we know that$\Gamma; \Delta \hookrightarrow \Gamma; !F$ and $\Gamma; !F \vdash F$ by (Ident), (!-Left), and (Derive).
By induction hypothesis, $\Gamma, \alpha; !\Delta' \vdash \overline{T_1} <: \overline{U_1}$. The result follows from (Sub Pos Rec *).

Case (Sub Refine Alg): $\Gamma; \Delta \vdash_{\mathsf{alg}} \overline{T} <: \overline{U}; F \otimes \forall y.(\mathit{forms}(y : T) \multimap \mathit{forms}(y : U))$ is proved by $\Gamma \vdash_{\mathsf{alg}} \psi(\overline{T}) <: \psi(\overline{U}); F$. We also know that $\Gamma; \Delta \vdash F \otimes \forall y.(\mathit{forms}(y : T) \multimap \mathit{forms}(y : U))$ and that at least one of the types $\overline{T}, \overline{U}$ is refined.

To show: $\Gamma; \Delta \vdash \overline{T} <:_{\mathsf{alg}} \overline{U}$ By (Rewrite), (Derive), and Lemma B.3 we know that $\Gamma; \Delta \hookrightarrow \Gamma; F, \forall y.(\mathit{forms}(y : T) \multimap \mathit{forms}(y : U))$, $\Gamma; F \vdash F$ and $\Gamma; \forall y.(\mathit{forms}(y : T) \multimap \mathit{forms}(y : U)) \vdash \forall y.(\mathit{forms}(y : T) \multimap \mathit{forms}(y : U))$ by (Ident) and (Derive).
By induction hypothesis, $\Gamma; F \vdash \psi(\overline{T}) <:_{\mathsf{alg}} \psi(\overline{U})$.
Without loss of generality, let us assume $y \notin \mathit{dom}(\Gamma)$ and, thus, $y \notin \mathit{fnfv}(\forall y.(\mathit{forms}(y : T) \multimap \mathit{forms}(y : U)))$. (This assumption can be fulfilled by α-renaming y if necessary.)
By Lemma B.3, we can easily see that $\Gamma, y : \psi(T); \forall y.(\mathit{forms}(y : T) \multimap \mathit{forms}(y : U)) \vdash \diamond$. By Lemma A.51, $\Gamma, y : \psi(T); \forall y.(\mathit{forms}(y : T) \multimap \mathit{forms}(y : U)) \vdash \mathit{forms}(y : T) \multimap \mathit{forms}(y : U)$. By Lemma A.50, $\Gamma, y : \psi(T); \Delta_2, \mathit{forms}(y : T) \vdash \mathit{forms}(y : U)$.
The result follows from (Sub Refine *).

We then prove part (1b). The proof proceeds by induction on the length of $\Gamma; \Delta \vdash \overline{T} <:_{\mathsf{alg}} \overline{U}$. The base cases (Sub Refl *) and (Sub Refl Rec *), follow by an inspection of the subtyping rules, by Lemma A.52, and by observing that $\Gamma; \Delta \vdash \mathbf{1}$ for any Δ such that $\Gamma; \Delta \vdash \diamond$. We now discuss the induction step by distinguishing the last applied intermediate subtyping rule.

Case (Sub Pub Tnt *): $\Gamma; \Delta \vdash T <:_{\mathsf{alg}} U$ is proved by $\Gamma; \Delta_1 \vdash T :: \mathsf{pub}$, $\Gamma; \Delta_2 \vdash U :: \mathsf{tnt}$, $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$, and $T \neq_{\mathrm{T}} U$. Note that here $T = \overline{T}$ and $U = \overline{U}$ are not annotated.

To show: $\Gamma \vdash_{\mathsf{alg}} T <: U; F_1 \otimes F_2$ By Lemma A.53, there exist F_1, F_2 such that $\Gamma \vdash_{\mathsf{alg}} T :: \mathsf{pub}; F_1$, $\Gamma \vdash_{\mathsf{alg}} U :: \mathsf{tnt}; F_2$, $\Gamma; \Delta_1 \vdash F_1$, and $\Gamma; \Delta_2 \vdash F_2$. The result follows from (Sub Pub Tnt Alg).

To show: $\Gamma; \Delta \vdash F_1 \otimes F_2$ By ($\otimes$-Right), $\Gamma; \Delta_1, \Delta_2 \vdash F_1 \otimes F_2$. The result follows from Lemma A.9.

Case (Sub Pub Tnt Rec *): The proof follows the same structure as the one for (Sub Pub Tnt *) and concludes with an application of (Sub Pub Tnt Rec Alg).

Case (Sub Fun *): $\Gamma; \Delta \vdash x : \overline{T_1} \to \overline{T_2} <:_{\mathsf{alg}} x : \overline{U_1} \to \overline{U_2}$ is proved by $\Gamma; !\Delta_1 \vdash \overline{U_1} <:_{\mathsf{alg}} \overline{T_1}$, $\Gamma, x : \psi(U_1); !\Delta_2 \vdash \overline{T_2} <:_{\mathsf{alg}} \overline{U_2}$, and $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2$. By induction hypothesis, there exist F_1, F_2 such that $\Gamma \vdash_{\mathsf{alg}} \overline{U_1} <: \overline{T_1}; F_1$, $\Gamma, x : \psi(U_1) \vdash_{\mathsf{alg}} \overline{T_2} <: \overline{U_2}; F_2$, $\Gamma; !\Delta_1 \vdash F_1$, and $\Gamma; !\Delta_2 \vdash F_2$.

To show: $\Gamma \vdash_{\mathsf{alg}} x : \overline{T_1} \to \overline{T_2} <: x : \overline{U_1} \to \overline{U_2}; !F_1 \otimes !F_2$ The result follows from (Sub Fun Alg).

To show: $\Gamma; \Delta \vdash !F_1 \otimes !F_2$ By (!-Right) and (Derive) and Lemma B.3 we know that $\Gamma; !\Delta_1 \vdash !F_1$ and $\Gamma; !\Delta_2 \vdash !F_2$. By ($\otimes$-Right) and (Derive), $\Gamma; !\Delta_1, !\Delta_2 \vdash !F_1 \otimes !F_2$. The result follows from Lemma A.9.

Case (Sub Pair *): The proof is analogous to the one for (Sub Fun *).

Case (Sub Sum *): The proof is analogous to the one for (Sub Fun *).

Case (Sub Pos Rec *): $\Gamma; \Delta \vdash \mu\alpha.\overline{T_1} <:_{\mathsf{alg}} \mu\alpha.\overline{U_1}$ is proved by $\Gamma, \alpha; !\Delta' \vdash \overline{T_1} <:_{\mathsf{alg}} \overline{U_1}$ and $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$. We furthermore know that α occurs only positively in $\overline{T_1}$ and $\overline{U_1}$. By induction hypothesis, there exists F such that $\Gamma, \alpha \vdash_{\mathsf{alg}} \overline{T_1} <: \overline{U_1}; F$ and $\Gamma; !\Delta' \vdash F$.

To show: $\Gamma \vdash_{\mathsf{alg}} \mu\alpha.\overline{T_1} <: \overline{U_1}; !F$ The result follows from (Sub Pos Rec Alg).

To show: $\Gamma; \Delta \vdash !F$ By (Derive) and (!-Right) we know that $\Gamma; !\Delta' \vdash !F$. We conclude using Lemma A.9.

Case (Sub Refine *): $\Gamma; \Delta \vdash \overline{T} <:_{\mathsf{alg}} \overline{U}$ is proved by $\Gamma; \Delta_1 \vdash \psi(\overline{T}) <:_{\mathsf{alg}} \psi(\overline{U})$, $\Gamma, y : \psi(T); \Delta_2, \mathit{forms}(y : T) \vdash \mathit{forms}(y : U)$, and $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$. By induction hypothesis, there exists F such that $\Gamma \vdash_{\mathsf{alg}} \psi(\overline{T}) <: \psi(\overline{U}); F$ and $\Gamma; \Delta_1 \vdash F$. We also know that at least one of the types $\overline{T}, \overline{U}$ is refined.

To show: $\Gamma; \Delta \vdash_{\mathsf{alg}} \overline{T} <: \overline{U}; (\forall y.\mathit{forms}(y : T) \multimap \mathit{forms}(y : U)) \otimes F$ The result follows by (Sub Refine Alg).

To show: $\Gamma; \Delta \vdash (\forall y.\mathit{forms}(y : T) \multimap \mathit{forms}(y : U)) \otimes F$ We know that $(\Gamma, y : \psi(T); \Delta_2, \mathit{forms}(y : T)) \vdash \mathit{forms}(y : U)$. By ($\multimap$-Right), $(\Gamma, y : \psi(T); \Delta_2) \vdash \mathit{forms}(y : T) \multimap \mathit{forms}(y : U)$.

By Lemma B.3, $\Gamma, x : \psi(T); \Delta_2 \vdash \diamond$ and, thus, $x \notin \mathit{dom}(\Gamma)$. Since $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$, we also have that $\Gamma; \Delta_1, \Delta_2 \vdash \diamond$ and, thus, $x \notin \mathit{fnfv}(\Delta_2) \subseteq \mathit{dom}(\Gamma)$.
By Lemma A.51, $(\Gamma; \Delta_2) \vdash \forall y.\mathit{forms}(y : T) \multimap \mathit{forms}(y : U)$. By ($\otimes$-Right), $\Gamma; \Delta_1, \Delta_2 \vdash F \otimes \forall y.(\mathit{forms}(y : T) \multimap \mathit{forms}(y : U))$. The result follows from Lemma A.9.

2. Part (2a) follows immediately from statement (1a) and the soundness of intermediate subtyping, shown in Lemma A.57, statement (3).

   Part (2b) follows immediately from statement (1b) and the completeness of intermediate subtyping, shown in Lemma A.57, statement (4).

□

The following lemma is used in the proof of soundness and completeness of algorithmic typing and states that for algorithmic subtyping type annotations need only occur in *either* the sub- or supertype.

Lemma A.59 (One-Sided Type annotations). *For all $\Gamma, \overline{T}, \overline{U}, T, UF$ such that $\Gamma \vdash_{\mathsf{alg}} \overline{T} <: \overline{U}; F$ such that $T = \langle\overline{T}\rangle$ and $U = \langle\overline{U}\rangle$ it holds that:*

1. *there exist $\overline{U}'$ such that $\langle\overline{U}\rangle = \langle\overline{U}'\rangle = U$ and $\Gamma \vdash_{\mathsf{alg}} T <: \overline{U}'; F$;*

2. *there exist $\overline{T}'$ such that $\langle\overline{T}\rangle = \langle\overline{T}'\rangle = T$ and $\Gamma \vdash_{\mathsf{alg}} \overline{T}' <: U; F$.*

Proof. The proof proceeds by simultaneous induction on the length of the derivation of $\Gamma \vdash_{\mathsf{alg}} \overline{T} <: \overline{U}; F$. For the base cases (Sub Refl Alg), (Sub Refl Rec Alg), and (Sub Pub Tnt Alg) we know that $\overline{T}, \overline{U}$ are not annotated and thus $\langle\overline{T}\rangle = \overline{T} = T$ and $\langle\overline{T}\rangle = \overline{T} = T$. We immediately conclude by selecting $\overline{T}' := T$ and $\overline{U}' := U$.

We now show the inductive cases:

Case (Sub Fun Alg) : Notice that the subtyping rule is contravariant in the input.

1. Statement (1) follows by applying the induction hypothesis (2) to the first subtyping premise and the induction hypothesis (1) to the second subtyping premise. We conclude by an application of (Sub Fun Alg).
2. Statement (2) follows by applying the induction hypothesis (1) to the first subtyping premise and the induction hypothesis (2) to the second subtyping premise. We conclude by an application of (Sub Fun Alg).

Case (Sub Pair Alg):

1. Statement (1) follows by applying the induction hypothesis (1) to the first subtyping premise and the induction hypothesis (1) to the second subtyping premise. We conclude by an application of (Sub Pair Alg).

2. Statement (2) follows by applying the induction hypothesis (2) to the first subtyping premise and the induction hypothesis (2) to the second subtyping premise. We conclude by an application of (Sub Pair Alg).

Case (Sub Sum Alg):

1. Statement (1) follows by applying the induction hypothesis (1) to the subtyping premise. We conclude by an application of (Sub Sum Alg).
2. Statement (2) follows by applying the induction hypothesis (2) to the subtyping premise. We conclude by an application of (Sub Sum Alg).

Case (Sub Pos Rec Alg):

1. Statement (1) follows by applying the induction hypothesis (1) to the subtyping premise. We conclude by an application of (Sub Pos Rec Alg).
2. Statement (2) follows by applying the induction hypothesis (2) to the subtyping premise. We conclude by an application of (Sub Pos Rec Alg).

Case (Sub Pub Tnt Rec Alg): In this case we know that $\overline{T} = (\mu\alpha.\, T')_s$ and $\overline{U} = (\mu\alpha.\, U')_{s'}$, where $s = \mathsf{SPT} \oplus s' = \mathsf{SPT}$. Furthermore, we know that $T = \langle\overline{T}\rangle = \mu\alpha.\, T'$ and $U = \langle\overline{U}\rangle = \mu\alpha.\, U'$ by definition of $\langle\bullet\rangle$.

1. We immediately conclude by selecting $\overline{U}' = (\mu\alpha.\, U')_{\mathsf{SPT}}$ and applying (Sub Pub Tnt Rec Alg).
2. We immediately conclude by selecting $\overline{T}' = (\mu\alpha.\, T')_{\mathsf{SPT}}$ and applying (Sub Pub Tnt Rec Alg).

Case (Sub Refine Alg): We observe that refinements types are never annotated on a top-level, only the core type stored therein may be.

1. Statement (1) follows by applying the induction hypothesis (1) to the subtyping premise and using Lemma A.55 and Lemma A.56. We conclude by an application of (Sub Refine Alg).
2. Statement (2) follows by applying the induction hypothesis (2) to the subtyping premise and using Lemma A.55 and Lemma A.56. We conclude by an application of (Sub Refine Alg).

□

The following proposition is used in the proof of soundness and completeness of algorithmic typing and states that extraction is unaffected by typing annotations.

Proposition A.2 (Annotated Extraction). *For all $\widetilde{a}, \Delta$ it holds that:*

1. *for all $\overline{E}, \overline{D}$ such that $\overline{E} \rightsquigarrow^{\widetilde{a}} [\Delta \mid \overline{D}]$ it must be the case that $\langle\overline{E}\rangle \rightsquigarrow^{\widetilde{a}} [\Delta \mid \langle\overline{D}\rangle]$;*
2. *for all $E, D, \overline{D}$ such that $D = \langle\overline{D}\rangle$ and $E \rightsquigarrow^{\widetilde{a}} [\Delta \mid D]$ it must be the case that there exists an annotated expression $\overline{E}$ such that $E = \langle\overline{E}\rangle$ and $\overline{E} \rightsquigarrow^{\widetilde{a}} [\Delta \mid \overline{D}]$.*

Lemma A.60 (Typing Truth Assumption)**.** *For all* Γ, Δ, T *such that* $\Gamma; \Delta \vdash \mathsf{assume}\ \mathbf{1} : T$ *it holds that* $\Gamma; \emptyset \vdash \mathsf{assume}\ \mathbf{1} : T$ *and* $\Gamma; \Delta \vdash \mathsf{unit} <: T$*.*

Proof. By induction on the length of the derivation $\Gamma; \Delta \vdash \mathsf{assume}\ \mathbf{1} : T$. We first note that $\Gamma; \emptyset \vdash \mathsf{assume}\ \mathbf{1} : \mathsf{unit}$ by (Exp True). By Lemma B.3 we know that $\Gamma; \Delta \vdash T$. In the base case (Exp True) we know that $T = \mathsf{unit}$ and conclude that $\Gamma; \Delta \vdash \mathsf{unit} <: \mathsf{unit}$ by (Sub Refl).

In the inductive case we know that the last applied rule must have been (Exp Subsum) and thus there exist T', Δ_1, Δ_2 such that $\Gamma; \Delta_1 \vdash \mathsf{assume}\ \mathbf{1} : T'$ and $\Gamma; \Delta_2 \vdash T' <: T$, where $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$. By induction hypothesis we know that $\Gamma; \Delta_1 \vdash \mathsf{unit} <: T'$ and we can thus conclude by an application of Lemma B.17 and Lemma A.9. □

Restatement A.3 (of Theorems 2.4 and 2.5)**.** *For all* Γ, Δ, T*, the following holds true:*

1. *for all* $\overline{E}, F$ *such that* $\Gamma \vdash_{\mathsf{alg}} \overline{E} : T; F$ *and* $\Gamma; \Delta \vdash F$*, we have that* $\Gamma; \Delta \vdash \langle \overline{E} \rangle : T$*;*

2. *for all* E *such that* $\Gamma; \Delta \vdash E : T$*, there exist* $\overline{E}, F$ *such that* $\langle \overline{E} \rangle = E$*,* $\Gamma \vdash_{\mathsf{alg}} \overline{E} : T; F$*, and* $\Gamma; \Delta \vdash F$*.*

Proof. 1. The proof proceeds by induction on the length of $\Gamma \vdash_{\mathsf{alg}} \overline{E} : T; F$. The base cases are (Val Var Alg), (Val Unit Alg), (Exp True Alg), (Exp Recv Alg), and (Exp Assert Alg): in all of these cases $\overline{E} = \langle \overline{E} \rangle$ (meaning $\overline{E}$ does not contain annotations) and they follow by an inspection of the typing rules and by Lemma A.52.

We show the induction cases in the following. Most cases follow a very similar structure so we show detailed examples for standard proof strategies and omit the details for analogous cases.

Case (Val Fun Alg): $\Gamma \vdash_{\mathsf{alg}} \lambda x : T_1. \overline{D} : x : T_1 \to T_2; !\forall x.(\mathit{forms}(x : T_1) \multimap F')$ is proved by $\Gamma, x : \psi(T_1) \vdash_{\mathsf{alg}} \overline{D} : T_2; F'$. We also know that $\Gamma; \Delta \vdash !\forall x.(\mathit{forms}(x : T_1) \multimap F')$.

To show: $\Gamma; \Delta \vdash \langle \lambda x : T. \overline{D} \rangle : x : T_1 \to T_2$ We first note that $\langle \lambda x : T. \overline{D} \rangle$ is equal to $\lambda x. \langle \overline{D} \rangle$. By (Rewrite), (Derive), and Lemma B.3 we know that $\Gamma; \Delta \hookrightarrow \Gamma; !\forall x.(\mathit{forms}(x : T_1) \multimap F')$ and $\Gamma; !\forall x.(\mathit{forms}(x : T_1) \multimap F') \vdash \forall x.(\mathit{forms}(x : T_1) \multimap F')$ by (Ident), (!-Left), and (Derive).
Without loss of generality, let us assume $x \notin \mathit{dom}(\Gamma)$ and, thus, $x \notin \mathit{fnfv}(!\forall x.(\mathit{forms}(x : T_1) \multimap F'))$. (This assumption can be fulfilled by α-renaming x if necessary.)
By Lemma B.3, we can easily see that $\Gamma, x : \psi(T_1); !\forall x.(\mathit{forms}(x : T_1) \multimap F') \vdash \diamond$. By Lemma A.51, $\Gamma, x : \psi(T_1); !\forall x.(\mathit{forms}(x : T_1) \multimap F') \vdash \mathit{forms}(x : T_1) \multimap F'$. By Lemma A.50, $\Gamma, x : \psi(T_1); !\forall x.(\mathit{forms}(x : T_1) \multimap F'), \mathit{forms}(x : T_1) \vdash F'$.

By induction hypothesis, $\Gamma, x : \psi(T_1); !\forall x.(\mathit{forms}(x : T_1) \multimap F'), \mathit{forms}(x : T_1) \vdash \langle\overline{D}\rangle : T_2$.

The result follows by an application of (Val Fun).

Case (Val Pair Alg): $\Gamma \vdash_{\mathsf{alg}} (\overline{M}, \overline{N}) : x : T_1 * T_2; !F_1 \otimes !F_2$ is proved by $\Gamma \vdash_{\mathsf{alg}} \overline{M} : T_1; F_1$ and $\Gamma \vdash_{\mathsf{alg}} \overline{N} : T_2\{M/x\}; F_2$, where $M := \langle\overline{M}\rangle$. We also know that $\Gamma; \Delta \vdash !F_1 \otimes !F_2$.

To show: $\Gamma; \Delta \vdash \langle(\overline{M}, \overline{N})\rangle : x : T_1 * T_2$ We first note that $\langle(\overline{M}, \overline{N})\rangle$ is equal to $(\langle\overline{M}\rangle, \langle\overline{N}\rangle)$. By (Rewrite), (Derive), and Lemma B.3 we know that $\Gamma; \Delta \hookrightarrow \Gamma; !F_1, !F_2$ and $\Gamma; !F_1 \vdash F_1$ and $\Gamma; !F_2 \vdash F_2$ by (Ident), (!-Left), and (Derive).

By applying the induction hypothesis twice we know that $\Gamma; !F_1 \vdash \langle\overline{M}\rangle : T_1$ and $\Gamma; !F_2 \vdash \langle\overline{N}\rangle : T_2\{\langle\overline{M}\rangle/x\}$.

The result follows by an application of (Val Pair).

Case (Val Inl Alg): $\Gamma \vdash_{\mathsf{alg}} (\mathsf{inl}\ \overline{M})_{\_+T_2} : T_1 + T_2; !F'$ is proved by $\Gamma \vdash_{\mathsf{alg}} \overline{M} : T_1; F_1$ and $\Gamma \vdash_{\mathsf{alg}} T_2$. We also know that $\Gamma; \Delta \vdash !F'$.

To show: $\Gamma; \Delta \vdash \langle(\mathsf{inl}\ \overline{M})_{\_+T_2}\rangle : T_1 + T_2$ We first note that $\langle(\mathsf{inl}\ \overline{M})_{\_+T_2}\rangle$ is equal to $\mathsf{inl}\ \langle\overline{M}\rangle$. By (Rewrite), (Derive), and Lemma B.3 we know that $\Gamma; \Delta \hookrightarrow \Gamma; !F'$ and $\Gamma; !F' \vdash F'$ by (Ident), (!-Left), and (Derive).

By applying the induction hypothesis we know that $\Gamma; !F' \vdash \langle\overline{M}\rangle : T_1$. By Lemma A.52 we know that $\Gamma; \emptyset \vdash T_2$ and thus by Lemma B.8 $\Gamma; !F' \vdash T_2$.

The result follows by an application of (Val Inl).

Case (Val Inr Alg): The proof is analogous to the one for (Val Inl Alg).

Case (Val Fold Alg): $\Gamma \vdash_{\mathsf{alg}} \mathsf{fold}\ \overline{M} : \mu\alpha.T'; !F'$ is proved by $\Gamma \vdash_{\mathsf{alg}} \overline{M} : T'\{\mu a.T'/\alpha\}; F'$. We also know that $\Gamma; \Delta \vdash !F'$.

To show: $\Gamma; \Delta \vdash \langle\mathsf{fold}\ \overline{M}\rangle : \mu\alpha.T'$ We first note that $\langle\mathsf{fold}\ \overline{M}\rangle$ is equal to $\mathsf{fold}\ \langle\overline{M}\rangle$. By (Rewrite), (Derive), and Lemma B.3 we know that $\Gamma; \Delta \hookrightarrow \Gamma; !F'$ and $\Gamma; !F' \vdash F'$ by (Ident), (!-Left), and (Derive).

By applying the induction hypothesis we know that $\Gamma; !F' \vdash \langle\overline{M}\rangle : T'\{\mu a.T'/\alpha\}$.

The result follows by an application of (Val Fold).

Case (Val Ref Alg): $\Gamma \vdash_{\mathsf{alg}} \overline{M}_{\{x:\_\ \mid\ F\}} : \{x : T' \mid F\}; F' \otimes F\{\langle\overline{M}\rangle/x\}$ is proved by $\Gamma \vdash_{\mathsf{alg}} \overline{M} : T'; F'$ and $\mathit{fnfv}(F) \subseteq \mathit{dom}(\Gamma) \cup \{x\}$. We also know that $\Gamma; \Delta \vdash F' \otimes F\{\langle\overline{M}\rangle/x\}$.

To show: $\Gamma; \Delta \vdash \langle\overline{M}_{\{x:\_\ \mid\ F\}}\rangle : \{x : T' \mid F\}$ We first note that $\langle\overline{M}_{\{x:\_\ \mid\ F\}}\rangle$ is equal to $\langle\overline{M}\rangle$. By (Rewrite), (Derive), and Lemma B.3 we know that $\Gamma; \Delta \hookrightarrow \Gamma; F', F\{\langle\overline{M}\rangle/x\}$ and $\Gamma; F' \vdash F'$ and $\Gamma; F\{\langle\overline{M}\rangle/x\} \vdash F\{\langle\overline{M}\rangle/x\}$ by (Ident). By induction hypothesis, $\Gamma; F' \vdash \langle\overline{M}\rangle : T$.

The result follows from (Val Refine).

Case (Exp Appl Alg): The proof follows straightforwardly from (Rewrite), (Derive), Lemma B.3, (Ident) by applying the induction hypothesis twice.

Case (Exp Let Alg): $\Gamma \vdash_{\mathsf{alg}} \mathsf{let}\ x = \overline{E_1}\ \mathsf{in}\ \overline{E_2} : T; \Delta' \multimap (F_1 \otimes (\forall x.\mathit{forms}(x : U) \multimap F_2))$ is proved by $\overline{E_1} \rightsquigarrow^{\emptyset} [\Delta' \mid \overline{E_1'}]$, $\Gamma \vdash_{\mathsf{alg}} \overline{E_1'} : U; F_1$, $\Gamma, x : \psi(U) \vdash_{\mathsf{alg}} \overline{E_2} : T; F_2$, $x \notin \mathit{fv}(T)$. We also know that $\Gamma; \Delta \vdash \Delta' \multimap (F_1 \otimes (\forall x.\mathit{forms}(x : U) \multimap F_2))$.

To show: $\Gamma; \Delta \vdash \langle \mathsf{let}\ x = \overline{E_1}\ \mathsf{in}\ \overline{E_2} \rangle : T$ We first note that $\langle \mathsf{let}\ x = \overline{E_1}\ \mathsf{in}\ \overline{E_2} \rangle$ is equal to $\mathsf{let}\ x = \langle \overline{E_1} \rangle\ \mathsf{in}\ \langle \overline{E_2} \rangle$. By Lemma A.50 and Lemma A.2, $\Gamma; \Delta, \Delta' \vdash F_1 \otimes (\forall x.\mathit{forms}(x : U) \multimap F_2)$.
By (Rewrite), (Derive), and Lemma B.3 it holds that $\Gamma; \Delta, \Delta' \hookrightarrow \Gamma; F_1, (\forall x.\mathit{forms}(x : U) \multimap F_2)$ and $\Gamma; F_1 \vdash F_1$ and $\Gamma; \forall x.\mathit{forms}(x : U) \multimap F_2 \vdash \forall x.\mathit{forms}(x : U) \multimap F_2$ by (Ident) and (Derive).
Without loss of generality, let us assume $x \notin \mathit{dom}(\Gamma)$ and, thus, $x \notin \mathit{fnfv}(\forall x.\mathit{forms}(x : U) \multimap F_2)$. (This assumption can be fulfilled by α-renaming x if necessary.)
By Lemma B.3, we can easily see that $\Gamma, x : \psi(U); \forall x.\mathit{forms}(x : U) \multimap F_2 \vdash \diamond$. By Lemma A.51, $\Gamma, x : \psi(U); \forall x.\mathit{forms}(x : U) \multimap F_2 \vdash \mathit{forms}(x : U) \multimap F_2$. By Lemma A.50, $\Gamma, x : \psi(U); \forall x.\mathit{forms}(x : U) \multimap F_2, \mathit{forms}(x : U) \vdash F_2$.
We note that by statement (1) of Proposition A.2 it holds that $\langle \overline{E_1} \rangle \rightsquigarrow^{\emptyset} [\Delta' \mid \langle \overline{E_1'} \rangle]$.
By induction hypothesis, $\Gamma; F_1 \vdash \langle \overline{E_1'} \rangle : T$ and $\Gamma, x : \psi(U); \forall x.\mathit{forms}(x : U) \multimap F_2, \mathit{forms}(x : U) \vdash \langle \overline{E_2} \rangle : U$.
The result follows from (Exp Let).

Case (Exp Split Alg): The proof follows a similar strategy as the one for (Exp Let Alg).

Case (Exp Match Alg): The proof follows a similar strategy as the one for (Exp Let Alg).

Case (Exp Eq Alg): The proof follows straightforwardly from (Rewrite), (Derive), (Ident), Lemma B.3, the induction hypothesis, ($\otimes$-Right), and Lemma A.9.

Case (Exp Assume Alg): $\Gamma \vdash_{\mathsf{alg}} (\mathsf{assume}\ F_1)_T : T; F_1 \multimap F_2$ is proved by $\Gamma \vdash_{\mathsf{alg}} (\mathsf{assume}\ \mathbf{1})_{\_<:T} : T; F_2$ and $\mathit{fnfv}(F) \subseteq \mathit{dom}(\Gamma)$, where $F_1 \neq \mathbf{1}$. We also know that $\Gamma; \Delta \vdash F_1 \multimap F_2$.

To show: $\Gamma; \Delta \vdash \langle (\mathsf{assume}\ F_1)_T \rangle : T$ We first note that $\langle (\mathsf{assume}\ F_1)_T \rangle$ is equal to $\mathsf{assume}\ F_1$.
By Lemma A.50 we know that $\Gamma; \Delta, F_1 \vdash F_2$.
By applying the induction hypothesis we know that $\Gamma; \Delta, F_1 \vdash \mathsf{assume}\ \mathbf{1} : T$.
The result follows by an application of (Exp Assume) .

Case (Exp Res Alg): The proof follows a similar and slightly simplified strategy as the one for (Exp Let Alg).

Case (Exp Send Alg): The proof follows straightforwardly from the induction hypothesis using the fact that $\langle a!\overline{M} \rangle$ is equal to $a!\langle \overline{M} \rangle$.

Case (Exp Fork Alg): The proof follows a similar strategy as the one for (Exp Let Alg).

2. The proof proceeds by induction on the length of $\Gamma; \Delta \vdash E : T$. The base cases are (Val Var), (Val Unit), (Exp True), (Exp Recv), and (Exp Assert): in these cases we choose $\overline{E} := E$ and $F := \mathbf{1}$. The statement follows by an inspection of the typing rules and by Lemma A.52.

   We show the induction cases in the following. Most cases follow a very similar structure so we show detailed examples for standard proof strategies and omit the details for analogous cases.

   For all cases the proof is split into two parts: we first show that there exists an annotated term $\overline{E}$ and a formula F such that $\Gamma \vdash_{\mathsf{alg}} \overline{E} : T; F$ and $\langle\overline{E}\rangle = E$. We then prove that $\Gamma; \Delta \vdash F$.

Case (Val Fun): $\Gamma; \Delta \vdash \lambda x. D : x : T_1 \to T_2$ is proved by $\Gamma, x : \psi(T_1); !\Delta', \mathit{forms}(x : T_1) \vdash D : T_2$ and $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$.

By induction hypothesis, there exist $\overline{D}, F'$ such that $\langle\overline{D}\rangle = D$, $\Gamma, x : \psi(T_1) \vdash \overline{D} : T_2; F'$ and $\Gamma, x : \psi(T_1); !\Delta', \mathit{forms}(x : T_1) \vdash F'$.

To show: $\Gamma \vdash_{\mathsf{alg}} \lambda x : T_1. \overline{D} : x : T_1 \to T_2; !\forall x.(\mathit{forms}(x : T_1) \multimap F')$ By Lemma B.3, $\mathit{fnfv}(T_1) \subseteq \mathit{dom}(\Gamma) \cup \{x\}$. The result follows from (Val Fun Alg). We note that $\langle\lambda x : T_1. \overline{D}\rangle = \lambda x. D$.

To show: $\Gamma; \Delta \vdash !\forall x.(\mathit{forms}(x : T_1) \multimap F')$ By ($\multimap$-Right), $\Gamma, x : \psi(T_1); !\Delta' \vdash \mathit{forms}(x : T_1) \multimap F'$.
By Lemma B.3, $\Gamma, x : \psi(T_1) \vdash \diamond$ and $x \notin \mathit{fnfv}(!\Delta') \subseteq \mathit{dom}(\Gamma)$. By Lemma A.51, $\Gamma; !\Delta' \vdash \forall x.(\mathit{forms}(x : T_1) \multimap F')$. By (!-Right), $\Gamma; !\Delta' \vdash !\forall x.(\mathit{forms}(x : T_1) \multimap F')$.
The result follows from Lemma A.9.

Case (Val Pair): $\Gamma; \Delta \vdash (M, N) : x : T_1 * T_2$ is proved by $\Gamma; !\Delta_1 \vdash M : T_1$ $\Gamma; !\Delta_2 \vdash N : T_2\{M/x\}$ and $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2$.

By applying the induction hypothesis twice we know that there exist $\overline{M}, \overline{N}, F_1, F_2$ such that $\langle\overline{M}\rangle = M$ and $\langle\overline{N}\rangle = N$ and $\Gamma \vdash \overline{M} : T_1; F_1$ and $\Gamma \vdash \overline{N} : T_2\{M/x\}; F_2$ and $\Gamma; !\Delta_1 \vdash F_1$ and $\Gamma; !\Delta_2 \vdash F_2$.

To show: $\Gamma \vdash_{\mathsf{alg}} (\overline{M}, \overline{N}) : x : T_1 * T_2; !F_1 \otimes !F_2$ The result follows immediately from (Val Pair Alg). We note that $\langle(\overline{M}, \overline{N})\rangle = (M, N)$.

To show: $\Gamma; \Delta \vdash !F_1 \otimes !F_2$ We apply (!-Right) to derive that $\Gamma; !\Delta_1 \vdash !F_1$ and $\Gamma; !\Delta_2 \vdash !F_2$. By ($\otimes$-Right), $\Gamma; !\Delta_1, !\Delta_2 \vdash !F_1 \otimes !F_2$.
The result follows from Lemma A.9.

Case (Val Inl): $\Gamma; \Delta \vdash \mathsf{inl}\ M : T_1 + T_2$ is proved by $\Gamma; !\Delta' \vdash M : T_1$ and $\Gamma; !\Delta' \vdash T_2$ and $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$.

By applying the induction hypothesis we know that there exist $\overline{M}, F'$ such that $\langle\overline{M}\rangle = M$ and $\Gamma \vdash \overline{M} : T_1; F'$ and $\Gamma; !\Delta' \vdash F'$.

To show: $\Gamma \vdash_{\mathsf{alg}} (\mathsf{inl}\ \overline{M})_{\_+T_2} : T_1 + T_2; !F'$ We know that $\Gamma; \emptyset \vdash_{\mathsf{alg}} T_2$ by Lemma B.3 and thus $\Gamma \vdash_{\mathsf{alg}} T_2$ by Lemma A.52. The result follows immediately from (VAL INL ALG). We note that $\langle(\mathsf{inl}\ \overline{M})_{\_+T_2}\rangle = \mathsf{inl}\ M$.

To show: $\Gamma; \Delta \vdash !F'$ By (!-RIGHT) we know that $\Gamma; !\Delta' \vdash !F'$. The result follows from Lemma A.9.

Case (VAL INR): The proof is analogous to the case of (VAL INL).

Case (VAL FOLD): $\Gamma; \Delta \vdash \mathsf{fold}\ M : \mu\alpha.T'$ is proved by $\Gamma; !\Delta' \vdash M : T'\{\mu\alpha.T'/\alpha\}$ and $\Gamma; \Delta \hookrightarrow \Gamma; !\Delta'$.

By applying the induction hypothesis we know that there exist $\overline{M}, F'$ such that $\langle\overline{M}\rangle = M$ and $\Gamma \vdash \overline{M} : T'\{\mu\alpha.T'/\alpha\}; F'$ and $\Gamma; !\Delta' \vdash F'$.

To show: $\Gamma \vdash_{\mathsf{alg}} \mathsf{fold}\ \overline{M} : \mu\alpha.T'; !F'$ We know that $\Gamma; \emptyset \vdash_{\mathsf{alg}} T_2$ by Lemma B.3 and thus $\Gamma \vdash_{\mathsf{alg}} T_2$ by Lemma A.52. The result follows immediately from (VAL INL ALG). We note that $\langle\mathsf{inl}\ \overline{M}\rangle = \mathsf{inl}\ M$.

To show: $\Gamma; \Delta \vdash !F'$ By (!-RIGHT) we know that $\Gamma; !\Delta' \vdash !F'$. The result follows from Lemma A.9.

Case (VAL REFINE): $\Gamma; \Delta \vdash M : \{x : T' \mid F'\}$ is proved by $\Gamma; \Delta_1 \vdash M : T'$, $\Gamma; \Delta_2 \vdash F'\{M/x\}$, and $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$.

By induction hypothesis, there exist $\overline{M}, F''$ such that $\langle\overline{M}\rangle = M$, $\Gamma \vdash_{\mathsf{alg}} \overline{M} : T'$, and $\Gamma; \Delta_1 \vdash F''$.

To show: $\Gamma \vdash_{\mathsf{alg}} \overline{M}_{\{x:\_ \mid F'\}} : \{x : T \mid F'\}; F'' \otimes F'\{M/x\}$ By Lemma B.3, $\mathit{fnfv}(F') \subseteq \mathit{dom}(\Gamma) \cup \{x\}$. The result follows from (VAL REF ALG). We note that $\langle\overline{M}_{\{x:\_ \mid F'\}}\rangle = M$.

To show: $\Gamma; \Delta \vdash F'' \otimes F'\{M/x\}$ The result follows from ($\otimes$-RIGHT) and Lemma A.9.

Case (EXP SUBSUM): $\Gamma; \Delta \vdash E : T$ is proved by $\Gamma; \Delta_1 \vdash E : T'$ and $\Gamma; \Delta_2 \vdash T' <: T$ and $\Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2$.

By applying the induction hypothesis we know that there exist $\overline{E}, F'$ such that $\langle\overline{E}\rangle = E$ and $\Gamma \vdash \overline{E} : T'; F'$ and $\Gamma; \Delta_1 \vdash F'$.

Furthermore, by Lemma B.23 we know that there exist $\overline{T'}, \overline{T}, F''$ such that that $T' = \langle\overline{T'}\rangle$, $T = \langle\overline{T}\rangle$, and $\Gamma \vdash_{\mathsf{alg}} \overline{T'} <: \overline{T}; F''$ and $\Gamma; \Delta_2 \vdash F''$. By Lemma A.59 it follows that there exists $\overline{T}^*$ such that that $T = \langle\overline{T}^*\rangle$ and $\Gamma \vdash_{\mathsf{alg}} T' <: \overline{T}^*; F''$

To show: $\Gamma \vdash_{\mathsf{alg}} \overline{E}_{\_<:\overline{T}^*} : T; F_1 \otimes F_2$ The result follows immediately from (EXP SUBSUM ALG). We note that $\langle\overline{E}\rangle = E$ as stated above.

To show: $\Gamma; \Delta \vdash !F'$ By (!-RIGHT) we know that $\Gamma; !\Delta' \vdash !F'$. The result follows from Lemma A.9.

Case (EXP APPL): The proof follows straightforwardly from the induction hypothesis using ($\otimes$-RIGHT) and Lemma A.9.

Case (EXP LET): $\Gamma; \Delta \vdash \mathsf{let}\ x = E_1\ \mathsf{in}\ E_2 : T$ is proved by $E_1 \rightsquigarrow^{\emptyset} [\Delta' \mid E_1']$, $\Gamma; \Delta_1 \vdash E_1' : U$, $\Gamma, x : \psi(U); \Delta_2, \mathit{forms}(x : U) \vdash E_2 : T$, $x \notin \mathit{fv}(T)$, and $\Gamma; \Delta, \Delta' \hookrightarrow \Gamma; \Delta_1, \Delta_2$.

By induction hypothesis, there exists $\overline{E'_1}, F_1$ such that $\langle\overline{E'_1}\rangle = E'_1$, $\Gamma \vdash_{\mathsf{alg}} \overline{E'_1} : U; F_1$, and $\Gamma; \Delta_1 \vdash F_1$. By induction hypothesis, there exists $\overline{E_2}, F_2$ such that $\langle\overline{E_2}\rangle = E_2, \Gamma, x : \psi(U) \vdash \overline{E_2} : T; F_2$, and $\Gamma, x : \psi(U); \Delta_2, \mathit{forms}(x : U) \vdash F_2$.

We note that by statement (2) of Proposition A.2 it holds that there exists $\overline{E_1}$ such that $\overline{E_1} \rightsquigarrow^{\emptyset} [\Delta' \mid \overline{E'_1}]$.

To show: $\Gamma \vdash_{\mathsf{alg}} \mathsf{let}\ x = \overline{E_1}\ \mathsf{in}\ \overline{E_2} : T; \Delta' \multimap (F_1 \otimes \forall x.(\mathit{forms}(x : U) \multimap F_2))$ By Lemma B.3, $\mathit{fnfv}(\Delta_1) \subseteq \mathit{dom}(\Gamma)$. The result follows from (Exp Let Alg). We note that $\langle \mathsf{let}\ x = \overline{E_1}\ \mathsf{in}\ \overline{E_2}\rangle$ is equal to $\mathsf{let}\ x = E_1\ \mathsf{in}\ E_2$.

To show: $\Gamma; \Delta \vdash \Delta' \multimap (F_1 \otimes \forall x.(\mathit{forms}(x : U) \multimap F_2))$ By ($\multimap$-Right), $\Gamma, x : \psi(U); \Delta_2 \vdash \mathit{forms}(x : U) \multimap F_2$. By Lemma B.3, $x \notin \mathit{fnfv}(\Delta_2) \subseteq \mathit{dom}(\Gamma)$. By Lemma A.51, $\Gamma, x : \psi(U); \Delta_2 \vdash \forall x.(\mathit{forms}(x : U) \multimap F_2)$. By ($\otimes$-Right), $\Gamma; \Delta_1, \Delta_2 \vdash F_1 \otimes \forall x.(\mathit{forms}(x : U) \multimap F_2)$. By Lemma A.9, $\Gamma; \Delta, \Delta' \vdash F_1 \otimes \forall x.(\mathit{forms}(x : U) \multimap F_2)$. By ($\multimap$-Right), $\Gamma; \Delta \vdash \Delta' \multimap (F_1 \otimes \forall x.(\mathit{forms}(x : U) \multimap F_2))$.

Case (Exp Split): The proof follows a similar strategy as the one for (Exp Let).

Case (Exp Match): The proof follows a similar strategy as the one for (Exp Let).

Case (Exp Eq): The proof follows straightforwardly from applying the induction hypothesis twice and using ($\otimes$-Right) and Lemma A.9.

Case (Exp Assume): $\Gamma; \Delta \vdash \mathsf{assume}\ F' : T$ is proved by $\Gamma; \Delta, F' \vdash \mathsf{assume}\ \mathbf{1} : T$, where $F' \neq \mathbf{1}$.

We first note that by Lemma A.60 it holds that $\Gamma; \emptyset \vdash \mathsf{assume}\ \mathbf{1} : \mathsf{unit}$ and $\Gamma; \Delta, F' \vdash \mathsf{unit} <: T$.

By combining Lemma B.23 and Lemma A.59 we know that there exist $\overline{T}, F''$ such that that $T = \langle\overline{T}\rangle$, and $\Gamma \vdash_{\mathsf{alg}} \mathsf{unit} <: \overline{T}; F''$ and $\Gamma; \Delta, F' \vdash F''$. By inspection of the algorithmic subtyping rules it follows that $\overline{T}$ must not contain any annotations ($T = \overline{T}$) and thus $\Gamma \vdash_{\mathsf{alg}} \mathsf{unit} <: T; F''$.

By applying the induction hypothesis (see proof of base case (Exp True)) to $\Gamma; \emptyset \vdash \mathsf{assume}\ \mathbf{1} : \mathsf{unit}$ it follows that $\Gamma \vdash_{\mathsf{alg}} \mathsf{assume}\ \mathbf{1} : \mathsf{unit}; \mathbf{1}$ and $\Gamma; \emptyset \vdash \mathbf{1}$.

To show: $\Gamma \vdash_{\mathsf{alg}} (\mathsf{assume}\ F')_T : T; F' \multimap (\mathbf{1} \otimes F'')$ We first apply (Exp Subsum Alg) to derive that $\Gamma \vdash_{\mathsf{alg}} (\mathsf{assume}\ \mathbf{1})\_{}_{<:T} : T; \mathbf{1} \otimes F''$.
We know that $\Gamma; \emptyset \vdash_{\mathsf{alg}} T_2$ by Lemma B.3 and thus $\Gamma \vdash_{\mathsf{alg}} T_2$ by Lemma A.52. The result follows from (Exp Assume Alg). We note that $\langle(\mathsf{assume}\ F')_T\rangle = \mathsf{assume}\ F'$.

To show: $\Gamma; \Delta \vdash F' \multimap (\mathbf{1} \otimes F'')$ As stated above we know that $\Gamma; \emptyset \vdash \mathbf{1}$ and $\Gamma; \Delta, F' \vdash F''$. By ($\otimes$-Right) it holds that $\Gamma; \Delta, F' \vdash \mathbf{1} \otimes F''$

Case (Exp Res): The proof follows a similar strategy as the one for (Exp Let).

Case (Exp Send): The proof follows straightforwardly from the induction hypothesis.

Case (Exp Fork): The proof follows a similar strategy as the one for (Exp Let).

□

(Sub Refl *)

$$\frac{\Gamma; \Delta \vdash T \qquad T \in \{\mathsf{unit}, \alpha\}}{\Gamma; \Delta \vdash T <:_{\mathsf{alg}} T}$$

(Sub Pub Tnt *)

$$\frac{\Gamma; \Delta_1 \vdash T :: \mathsf{pub} \qquad \Gamma; \Delta_2 \vdash U :: \mathsf{tnt} \qquad \Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2 \qquad T \neq_{\top} U}{\Gamma; \Delta \vdash T <:_{\mathsf{alg}} U}$$

(Sub Fun *)

$$\frac{\begin{array}{c}\Gamma; !\Delta_1 \vdash \overline{T'} <:_{\mathsf{alg}} \overline{T} \\ \Gamma, x : \psi(T'); !\Delta_2 \vdash \overline{U} <:_{\mathsf{alg}} \overline{U'} \\ \Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2\end{array}}{\Gamma; \Delta \vdash \overline{x : \overline{T} \rightarrow \overline{U}} <:_{\mathsf{alg}} \overline{x : \overline{T'} \rightarrow \overline{U'}}}$$

(Sub Pair *)

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

(Sub Sum *)

$$\frac{\begin{array}{c}\Gamma; !\Delta_1 \vdash \overline{T} <:_{\mathsf{alg}} \overline{T'} \qquad \Gamma; !\Delta_2 \vdash \overline{U} <:_{\mathsf{alg}} \overline{U'} \\ \Gamma; \Delta \hookrightarrow \Gamma; !\Delta_1, !\Delta_2\end{array}}{\Gamma; \Delta \vdash \overline{\overline{T} + \overline{U}} <:_{\mathsf{alg}} \overline{\overline{T'} + \overline{U'}}}$$

(Sub Pos Rec *)

$$\frac{\begin{array}{c}\Gamma, \alpha; !\Delta' \vdash \overline{T} <:_{\mathsf{alg}} \overline{T'} \\ \alpha \text{ occurs only positively in } \overline{T} \text{ and } \overline{T'} \\ T \neq T' \qquad \Gamma; \Delta \hookrightarrow \Gamma; !\Delta'\end{array}}{\Gamma; \Delta \vdash \mu\alpha. \overline{T} <:_{\mathsf{alg}} \mu\alpha. \overline{T'}}$$

(Sub Refl Rec *)

$$\frac{\Gamma; \Delta \vdash \mu\alpha. T}{\Gamma; \Delta \vdash \mu\alpha. T <:_{\mathsf{alg}} \mu\alpha. T}$$

(Sub Pub Tnt Rec *)

$$\frac{\begin{array}{c}\Gamma; \Delta_1 \vdash \mu\alpha. T :: \mathsf{pub} \qquad \Gamma; \Delta_2 \vdash \mu\alpha. U :: \mathsf{tnt} \\ s = \mathsf{SPT} \oplus s' = \mathsf{SPT} \qquad \Gamma; \Delta \hookrightarrow \Gamma; \Delta_1, \Delta_2\end{array}}{\Gamma; \Delta \vdash (\mu\alpha. T)_s <:_{\mathsf{alg}} (\mu\alpha. U)_{s'}}$$

(Sub Refine *)

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

Notation: We use T to denote the non-annotated counterpart $\langle\overline{T}\rangle$ of the annotated type $\overline{T}$.

Table A.2: Intermediate subtyping relation $<:_{\mathsf{alg}}$ (AF7)

B

Proofs of Chapter 4

B.1 Soundness of DF7

In the following section we prove the the soundness of the DF7 type system presented in Chapter 4.

We first show some basic properties of DF7. We then introduce some assumptions on the signature, before explaining the notion of distance of expressions. This allows us to then prove the type and distance preservation theorem needed to show the final differential privacy results of our approach.

B.1.1 Basic properties

To simplify notation we let $\mathcal{J}$ denote judgements that range over $\{\diamond, \tau, \theta :: \kappa, \tau <: \rho, A : \tau\}$.

Proposition B.1 (Adding Environments Preserves Ordering). *If* $\Gamma = \Gamma_1 + \Gamma_2$, $\Gamma' = \Gamma'_1 + \Gamma'_2$ *and* $\Gamma'' = \Gamma''_1 + \Gamma''_2$, *then* $\Gamma, \Gamma', \Gamma'' = \Gamma_1, \Gamma'_1, \Gamma''_1 + \Gamma_2, \Gamma'_2, \Gamma''_2$.

Proposition B.2 (Domains of Added Environments). *If* $\Gamma = \Gamma_1 + \Gamma_2$, *then* $dom(\Gamma) = dom(\Gamma_1) = dom(\Gamma_2)$.

Lemma B.1 (Pruning Preserves Well-formation). *If* $\Gamma, \Gamma' \vdash \diamond$, *then* $\Gamma \vdash \diamond$.

Proof. By induction on the structure of Γ'. □

Lemma B.2 (Well-formation of Added Environments). *The following statements hold:*

1. *if* $\Gamma \vdash \diamond$ *and* $\Gamma = \Gamma_1 + \Gamma_2$, *then* $\Gamma_1 \vdash \diamond$ *and* $\Gamma_2 \vdash \diamond$;

2. *if* $\Gamma = \Gamma_1 + \Gamma_2$, $\Gamma_1 \vdash \diamond$ *and* $\Gamma_2 \vdash \diamond$, *then* $\Gamma \vdash \diamond$.

Proof. Point 1 is proved by induction on the derivation of $\Gamma \vdash \diamond$ using Proposition B.2, while point 2 is proved by induction on the derivation of $\Gamma = \Gamma_1 + \Gamma_2$, using Proposition B.2 and Lemma B.1. □

Lemma B.3 (Derived Judgments)**.** *The following statements hold:*

1. *if* $\Gamma \vdash \theta$*, then* $\Gamma \vdash \diamond$*,* $\mathit{ft}(\theta) \subseteq \mathit{dom}(\Gamma)$*;*
2. *if* $\Gamma \vdash \theta :: \kappa$*, then* $\Gamma \vdash \theta$*;*
3. *if* $\Gamma \vdash \tau <: \rho$*, then* $\Gamma \vdash \tau$ *and* $\Gamma \vdash \rho$*;*
4. *if* $\Gamma \vdash A : \tau$*, then* $\Gamma \vdash \tau$ *and* $\mathit{ft}(A) \cup \mathit{fv}(A) \subseteq \mathit{dom}(\Gamma)$*.*

Proof. We first observe that for every core type ϕ it holds that $\mathit{ft}(\phi) = \mathit{ft}(!_k\phi)$. The first statement follows immediately by the rule Type. The last three statements are proven in the order that they are stated by induction on the depth of the derivation of the judgment using the previous results as well as Lemma B.1 and Lemma B.2. □

Lemma B.4 (Exchange)**.** *If* $\Gamma, \mu_1, \mu_2, \Gamma' \vdash \mathcal{J}$ *and* $\mathit{dom}(\mu_1) \cap \mathit{ft}(\mu_2) = \emptyset$*, then* $\Gamma, \mu_2, \mu_1, \Gamma' \vdash \mathcal{J}$*.*

Proof. By induction on the depth of the derivation of $\Gamma, \mu_1, \mu_2, \Gamma' \vdash \mathcal{J}$. □

Lemma B.5 (Type Variable Strengthening)**.** *For some environments* Γ, Γ' *and some entry* $\mu \in \{\alpha, \alpha :: \kappa\}$ *for some type variable* α*, where* $\mathit{dom}(\mu) \cap \mathit{ft}(\Gamma') = \emptyset$ *it holds that*

- *if* $\Gamma, \mu, \Gamma' \vdash \diamond$*, then* $\Gamma, \Gamma' \vdash \diamond$*;*
- *if* $\Gamma, \mu, \Gamma' \vdash \theta :: \kappa$ *and* $\mathit{dom}(\mu) \cap \mathit{ft}(\theta) = \emptyset$ *, then* $\Gamma, \Gamma' \vdash \theta :: \kappa$*;*
- *if* $\Gamma, \mu, \Gamma' \vdash \tau <: \rho$ *and* $\mathit{dom}(\mu) \cap \mathit{ft}(\tau, \rho) = \emptyset$*, then* $\Gamma, \Gamma' \vdash \tau <: \rho$*.*

Proof. The statements are proven by induction on the derivation of $\Gamma, \mu, \Gamma' \vdash \diamond$in the order that they are stated, using previous results for the later ones. □

Lemma B.6 (Variable Binding Strengthening)**.** *For some environments* Γ, Γ' *and some type binding of the form* $x : \tau$ *and* $x :!_0\phi$*, core types* ϕ*, types* ρ, ρ'*, (core) types* θ*, and kinds* κ *it holds that*

1. *if* $\Gamma, x : \tau, \Gamma' \vdash \diamond$*, then* $\Gamma, \Gamma' \vdash \diamond$*;*
2. *if* $\Gamma, x : \tau, \Gamma' \vdash \theta$*, then* $\Gamma, \Gamma' \vdash \theta$*;*
3. *if* $\Gamma, x : \tau, \Gamma' \vdash \theta :: \kappa$*, then* $\Gamma, \Gamma' \vdash \theta :: \kappa$*;*
4. *if* $\Gamma, x : \tau, \Gamma' \vdash \rho <: \rho'$*, then* $\Gamma, \Gamma' \vdash \rho <: \rho'$*;*
5. *if* $\Gamma, x :!_0\phi, \Gamma' \vdash A : \rho$*, then* $\Gamma, \Gamma' \vdash A : \rho$*.*

Proof. The first four statement are proven in order by induction on the derivation of $\Gamma, x : \tau, \Gamma' \vdash \mathcal{J}$, using the previous results. Intuitively, subtyping and kinding rules never consider variable bindings. The last statement is proven by induction on the derivation of $\Gamma, x :!_0\phi, \Gamma' \vdash A : \rho$. Note that the rules VAR and !I both require an index > 0, meaning $x :!_0\phi$ is never used. □

Definition B.1 (Exponential Fragment). *For all environments* Γ *we define the* exponential fragment *of* Γ *to be an exponential environment* Γ' *such that* $\Gamma = \Gamma + \Gamma'$.

Note that there might exist multiple exponential fragments of the same environment.

Lemma B.7 (Properties of Exponential Fragments). *For all environments* Γ *and all exponential fragments* Γ' *of* Γ *the following properties hold:*

1. *if* $\Gamma \vdash \mathcal{J}$*, where* $\mathcal{J} \in \{\diamond, \theta, \theta :: \kappa, \tau <: \rho\}$ *then* $\Gamma' \vdash \mathcal{J}$.
2. *for all environments* Γ'' *such that* $\Gamma + \Gamma'' \vdash \diamond$ *it holds that* Γ' *is an exponential fragment of* $\Gamma + \Gamma''$.
3. *if* Δ' *is an exponential fragment of* Γ *then* $\Gamma' + \Delta'$ *is also an exponential fragment of* Γ.

Proof. 1. By induction on the derivation of $\Gamma \vdash \mathcal{J}$, using the definition of environment sum, Proposition B.2, Lemma B.2, and Lemma B.6.

2. Follows immediately by Definition B.1 and Proposition B.2.

3. Follows immediately by Definition B.1 and Proposition B.2.

□

Lemma B.8 (Weakening). *For all environments* Γ, Γ'*, environment entries* μ*, and judgements* $\mathcal{J}$ *it holds that if* $\Gamma, \Gamma' \vdash \mathcal{J}$ *and* $\Gamma, \mu, \Gamma' \vdash \diamond$ *then* $\Gamma, \mu, \Gamma' \vdash \mathcal{J}$.

Proof. By induction on the derivation of $\Gamma, \Gamma' \vdash \mathcal{J}$ using the fact that

$$dom(\Gamma, \mu, \Gamma') \supseteq dom(\Gamma, \Gamma')$$

. □

Lemma B.9 (Weakening by Environment Adding). *It holds that:*

1. *For all environments* Γ*, environment entries* μ*, and judgements* $\mathcal{J}$ *it holds that if* $\Gamma \vdash \mathcal{J}$ *and* $\Gamma + \mu \vdash \diamond$ *then* $\Gamma + \mu \vdash \mathcal{J}$.
2. *For all environments* Γ, Γ' *and judgements* $\mathcal{J}$ *it holds that if* $\Gamma \vdash \mathcal{J}$ *and* $\Gamma + \Gamma' \vdash \diamond$ *then* $\Gamma + \Gamma' \vdash \mathcal{J}$.

Proof. 1. By induction on the derivation of $\Gamma \vdash \mathcal{J}$.

2. By induction on the size of Γ' by applying the first property multiple times. □

Lemma B.10 (Bound Weakening). *For all environments $\Gamma, \Gamma', \Gamma''$, all types τ, ρ, and all jugements $\mathcal{J}$ such that Γ' is an exponential fragment of Γ such that $\Gamma' \vdash \tau <: \rho$ and $\Gamma, x : \rho, \Gamma'' \vdash \mathcal{J}$ it holds that $\Gamma, x : \tau, \Gamma'' \vdash \mathcal{J}$ and for all non-typing judgements $\mathcal{J}$ the depth of the second derivation equals that of the first.*

Proof. By induction on the derivation of $\Gamma, x : \rho, \Gamma'' \vdash \mathcal{J}$. As we have noticed before, kinding and subtyping are not affected by variable bindings at all. For typing judgements the only interesting case is VAR which is extended by an application of SUB if necessary. □

Lemma B.11 (Typing Replicated Terms). *For all environments Γ, terms M, reals $k > 0$, and types τ it holds that if $\Gamma \vdash M : \tau$ then*

1. *$k\Gamma \vdash M :!_k\tau$ and*
2. *$k\Gamma \vdash M :!_{k'}\tau$ for all k' such that $1 \leq k' \leq k$.*

Proof. 1. In the case of $k = 1$ there is nothing to prove. For $k \neq 1$ it follows immediately by $!_k$I.

2. Follows by the previous statement and Lemma B.9. □

Lemma B.12 (Substitution). *For all environments $\Gamma_1, \Gamma_2, \Gamma_3$, terms M, expressions B, and types τ, τ' for which it holds that $\Gamma_1 \vdash M : \tau$ and $\Gamma_2, x :!_k\tau, \Gamma_3 \vdash B : \tau'$ for some k it follows that $k\Gamma_1 + (\Gamma_2, \Gamma_3) \vdash B\{M/x\} : \tau'$.*

Proof. By induction on the derivation of $\Gamma_2, x :!_k\tau, \Gamma_3 \vdash B : \tau'$. We use the fact that by Lemma B.3 $\Gamma_1 \vdash \diamond$ and that by the same Lemma together with Lemma B.6 it must be the case that $\Gamma_2, \Gamma_3 \vdash \diamond$ and thus by Lemma B.2 $k\Gamma_1 + (\Gamma_2, \Gamma_3) \vdash \diamond$. We also note that $k\Gamma_1 + (\Gamma_2, \Gamma_3) \vdash \tau'$ by the definition of TYPE, Proposition B.2, and Lemma B.3. This immediately takes care of the base cases SIG, READ, READ OPP, WRITE, REF, and REF OPP. In the base case VAR it must be the case that $B = y$ for some variable y. We distinguish whether $y \neq x$ or $x = y$. In the first case we know that $B\{M/x\} = y\{M/x\} = y$ and that $y : \tau' \in \Gamma_2, \Gamma_3$. We can apply rule VAR to derive that $\Gamma_2, \Gamma_3 \vdash y : \tau'$ and then conclude by Lemma B.9. In the case that $B = x$ we need to show that $r\Gamma_1 + (\Gamma_2, \Gamma_3) \vdash M : \tau'$.

By the definition of VAR we know that $x : \tau' \in \Gamma_2, x :!_k\tau, \Gamma_3$ and thus $\tau' =!_k\tau$. Thus, we need to show that $k\Gamma_1 + (\Gamma_2, \Gamma_3) \vdash M :!_k\tau$, which follows immediately by application of Lemma B.11 and Lemma B.9.

The inductive cases $+/\mu$I, $!_k$I, $\multimap$I, ADD-NOISE, EXP-MECH, and SAN follow all similarly by applying the induction hypothesis to the hypotheses of the applied rule and then applying the same rule again on the result.

Case $\otimes$I is of the form

$$\frac{\Delta_1 \vdash M_1 : \tau_2 \qquad \Delta_2 \vdash M_2 : \tau_2}{\Gamma_2, x :!_k\tau, \Gamma_3 \vdash (M_1, M_2) : \underbrace{!_1(\tau_1 \otimes \tau_2)}_{\tau'}}$$

for some $M_1, M_2, \tau_1, \tau_2, \Delta_1, \Delta_2$ such that $\Gamma_2, x :!_k\tau, \Gamma_3 = \Delta_1 + \Delta_2$. We distinguish three cases: either there exists k', such that $0 < k' < k$ and $x :!_k\tau \in \Delta_1$ and $x :!_{k'-k}\tau \in \Delta_2$ or $x :!_k\tau \in \Delta_1$ and $x; !_0\tau \in \Delta_2$ or $x :!_0\tau \in \Delta_1$ and $x :!_k\tau \in \Delta_2$. In the first case we apply the induction hypothesis to both premises and conclude by applying ⊗I. The last two cases are symmetric so without loss of generality we assume that $x :!_k\tau \in \Delta_1$ and $x :!_0\tau \in \Delta_2$. By Lemma B.6 we know that the latter implies that $\Delta'_2 \vdash M_2 : \tau_2$, where $\Delta'_2 := \Delta_2 \setminus x :!_0\tau$, thus $x \notin \mathit{dom}(\Delta'_2)$. We apply the induction hypothesis to the first hypothesis. Furthermore, we know by Lemma B.3 that $x \notin \mathit{fv}(M_2)$ and that thus $M_2\{M/x\} = M_2$. We can conclude by applying ⊗I. The remaining cases ⊗E, +E, μE, ⊸E, Let, and Equal are treated very similarly to the previous one.

In the case Sub we apply the induction hypothesis and Lemma B.6 together with Lemma B.9 and conclude by applying Sub.

□

Definition B.2 (Type Substitutions). *For convenience we define the following notation:*

- $\Gamma\{\tau/\alpha\} = \mu_1\{\tau/\alpha\}, \ldots, \mu_n\{\tau/\alpha\}$, *where* $\alpha \notin \mathit{dom}(\Gamma)$ *and* $\Gamma = \mu_1, \ldots, \mu_n$
- $\mu\{\tau/\alpha\} = \begin{cases} y : \tau'\{\tau/\alpha\} & \textit{if } \mu = y : \tau' \\ \mu & \textit{otherwise} \end{cases}$
- $A\{\tau/\alpha\}$ *is recursively defined. We just show the base cases, the inductive cases are standard.*
$$A\{\tau/\alpha\} = \begin{cases} \mathsf{ref}_{\tau'\{\tau/\alpha\}} & \textit{if } A = \mathsf{ref}_{\tau'} \\ \mathsf{read}_{a:\tau'\{\tau/\alpha\}} & \textit{if } A = \mathsf{read}_{a:\tau'} \\ \mathsf{write}_{a:\tau'\{\tau/\alpha\}} & \textit{if } A = \mathsf{write}_{a:\tau'} \\ A & \textit{if } A \in \{x, c, f\} \end{cases}$$
- *Using this we define* $(A : \tau')\{\tau/\alpha\} = A\{\tau/\alpha\} : \tau'\{\tau/\alpha\}$.

Lemma B.13 (Core Type Substitution). *It holds that:*

1. *For all environments* Γ, Γ', *type variables* α, *core types* ϕ *and judgements* $\mathcal{J}$ *such that* $\Gamma, \alpha, \Gamma' \vdash \mathcal{J}$ *and* $\Gamma \vdash \phi$ *it holds that* $\Gamma, \Gamma'\{\phi/\alpha\} \vdash \mathcal{J}\{\phi/\alpha\}$.
2. *For all environments* Γ, Γ', *type variables* α, *core types* ϕ *and kinds* κ *such that* $\Gamma, \alpha :: \kappa, \Gamma' \vdash \diamond$ *and* $\Gamma \vdash \phi :: \kappa$ *it holds that* $\Gamma, \Gamma'\{\phi/\alpha\} \vdash \diamond$.
3. *For all environments* Γ, Γ', *type variables* α, *core types* ϕ, *kinds* κ, *and (core) types* θ *such that* $\Gamma, \alpha :: \kappa, \Gamma' \vdash \theta$ *and* $\Gamma \vdash \phi :: \kappa$ *it holds that* $\Gamma, \Gamma'\{\phi/\alpha\} \vdash \theta\{\phi/\alpha\}$.
4. *For all environments* Γ, Γ', *type variables* α, *core types* ϕ *and kinds* κ, κ', *and (core) types* θ *such that* $\Gamma, \alpha :: \kappa, \Gamma' \vdash \theta :: \kappa'$ *and* $\Gamma \vdash \phi :: \kappa$ *it holds that* $\Gamma, \Gamma'\{\phi/\alpha\} \vdash \theta\{\phi/\alpha\} :: \kappa'$.

Proof. 1. Individual proof by induction on the derivation of $\Gamma, \alpha, \Gamma' \vdash \mathcal{J}$ for each judgement $\mathcal{J}$.

2. By induction on the derivation of $\Gamma, \alpha :: \kappa, \Gamma' \vdash \theta$ using Lemma B.3.

3. By rule Type, point (1), and Lemma B.3.

4. By induction on the derivation of $\Gamma, \alpha :: \kappa, \Gamma' \vdash \theta :: \kappa'$.

□

Lemma B.14 (Public Down - Tainted Up). *For all environments Γ and types τ, ρ it holds that*

1. *if $\Gamma \vdash \tau <: \rho$ and $\Gamma \vdash \rho :: \mathsf{pub}$ then $\Gamma \vdash \tau :: \mathsf{pub}$;*

2. *if $\Gamma \vdash \tau <: \rho$ and $\Gamma \vdash \tau :: \mathsf{tnt}$ then $\Gamma \vdash \rho :: \mathsf{tnt}$.*

Proof. Both statements are proven simultaneously by induction on the derivation of $\Gamma \vdash \tau <: \rho$. □

Lemma B.15 ($\mathcal{OPP}$ Public Tainted). *For all opponent types $\tau \in \mathcal{OPP}$ it holds that $\emptyset \vdash \tau :: \kappa$ for $\kappa \in \{\mathsf{tnt}, \mathsf{pub}\}$.*

Proof. By induction on the structure of τ. We note that by definition of $\mathcal{OPP}$ all replication indexes in τ must be set to ∞, so $\tau = [\tau]_\infty$. □

Lemma B.16 (Public Tainted). *For all environments Γ and types τ it holds that*

1. *$\Gamma \vdash \tau :: \mathsf{pub}$ if and only if $\Gamma \vdash \tau <: \rho$, for all $\rho \in \mathcal{OPP}$;*

2. *$\Gamma \vdash \tau :: \mathsf{tnt}$ if and only if $\Gamma \vdash \rho <: \tau$, for all $\rho \in \mathcal{OPP}$.*

Proof. We first note that by Lemma B.15 $\emptyset \vdash \rho :: \kappa$ for $\kappa \in \{\mathsf{tnt}, \mathsf{pub}\}$. The forward implication thus follows immediately by application of rule Sub Kind. The inverse implication follows by Lemma B.14. □

Lemma B.17 (Transitivity). *For all environments Γ and types τ, ρ, τ' it holds that if $\Gamma \vdash \tau <: \rho$ and $\Gamma \vdash \rho <: \tau'$ then $\Gamma \vdash \tau <: \tau'$.*

Proof. By induction on the sum of the depth of the derivations of $\Gamma \vdash \tau <: \rho$ and $\Gamma \vdash \rho <: \tau'$. We denote the last applied rule of the derivation $\Gamma \vdash \tau <: \rho$ by R_1 and the last applied rule of the derivation $\Gamma \vdash \rho <: \tau'$ by R_2 and proceed by case analysis of the applied rules (R_1, R_2).

Case R_1 and/or R_2 are Sub Refl: We immediately conclude by Lemma B.9.

Case R_1 is Sub Kind: In this case we know that $\Gamma \vdash \tau :: \mathsf{pub}$ and $\Gamma \vdash \rho :: \mathsf{tnt}$. We can apply Lemma B.14 to derive that $\Gamma \vdash \tau' :: \mathsf{tnt}$. We immediately conclude by applying Sub Kind.

Case R_2 is Sub Kind: In this case we know that $\Gamma \vdash \rho :: \mathsf{pub}$ and $\Gamma \vdash \tau' :: \mathsf{tnt}$. We can apply Lemma B.14 to derive that $\Gamma \vdash \tau :: \mathsf{pub}$. We immediately conclude by applying Sub Kind.

Case $R_1 = R_2$: For all cases in which neither R_1 nor R_2 are equal to Sub Refl or Sub Kind all types τ, τ', ρ have the same structure and we can thus conclude by applying the induction hypothesis whenever necessary.

Case No other case can be possible.

□

Lemma B.18 (Inversion). *It holds that:*

1. *If* $\Gamma \vdash !_1\phi <: !_1\psi$ *then the last applied subtyping rule in that derivation must not be* Sub Kind.
2. *For all environments* Γ*, types* τ*, and expressions* A *it holds that if* $\Gamma \vdash A : \tau$ *then either the last applied rule of the derivation* $\Gamma \vdash A : \tau$ *is not equal to* Sub *or the last applied rule of the derivation is* Sub *and there exists a typing rule* R *different from* Sub *and a type* ρ *such that* $\Gamma \vdash A : \rho$*, where the last applied typing rule in that derivation is* R *and* $\Gamma \vdash \rho <: \tau$.
3. *For all environments* Γ*, types* τ*, and expressions* A *it holds that if* $\Gamma \vdash A : \tau$ *there exists a type* ρ *such that the derivation* $\Gamma \vdash A : \rho$ *ends with the application of a rule that is not equal to* Sub *and* $\Gamma \vdash \rho <: \tau$.

Proof. 1. By inspection of the kinding rules we see that $\Gamma \vdash !_1\phi :: \mathsf{pub}$ can never be fulfilled, thus the premises for applying Sub Kind are not fulfilled.

2. By induction on the derivation of $\Gamma \vdash A : \tau$. We proceed by case distinction on the last applied rule R. The only interesting case is R equal to Sub. In this case we know that $\Gamma \vdash A : \tau'$ for some τ' and $\Gamma \vdash \tau' <: \tau$. If the last applied rule in the derivation of $\Gamma \vdash A : \tau'$ is different from Sub we can conclude immediately. Otherwise, by an application of the induction hypothesis we can derive that there exists a ρ such that $\Gamma \vdash \rho <: \tau'$ and $\Gamma \vdash A : \rho'$ with a last applied rule different from Sub. We conclude by applying Lemma B.17 by which $\Gamma \vdash \rho <: \tau$.

3. Follows immediately from the previous statement and Sub Refl.

□

Lemma B.19 (Inequalities for Fractions). *For all* $n \in \mathbb{N}^{>0}$ *and all* $a \in \mathbb{R}$ *and all* $c_j, d_j \in \mathbb{R}^{\geq 0}$ *it holds that if* $\frac{c_j}{d_j} \leq a$ *for all* $j \in [1, n]$ *then* $\frac{\sum_{i=1}^n c_i}{\sum_{i=1}^n d_i} \leq a$.

Proof. By induction on n.

Case $n = 1$: Follows immediately.

Case $n = 2$: We first note that for $m := \mathsf{max}_{i \in \{1,2\}}(\frac{c_i}{d_i})$ it holds that $m \leq a$. This follows immediately since otherwise we would contradict the assumption that $\frac{c_i}{d_i} \leq a$ for all $i \in \{1, 2\}$. It thus suffices to show that $\frac{c_1+c_2}{d_1+d_2} \leq m$. Note that $\frac{c_i}{d_i} \leq m \Leftrightarrow c_i \leq m \cdot d_i$ for $i \in \{1, 2\}$, since $d_i \geq 0$ by definition. We conclude as follows:

$$\frac{c_1 + c_2}{d_1 + d_2} \leq \frac{m \cdot d_1 + m \cdot d_2}{d_1 + d_2} = \frac{m \cdot (d_1 + d_2)}{d_1 + d_2} = m.$$

Case $n \geq 2$: We know that $\frac{c_n}{d_n} \leq a$ and that by applying the induction hypothesis for $n-1$ we can derive that $\frac{\sum_{i=1}^{n-1} c_i}{\sum_{i=1}^{n-1} d_i} \leq a$. Since this number must be ≥ 0 we can thus apply the induction hypothesis for 2 to derive that $\frac{(\sum_{i=1}^{n} c_i)+c_n}{(\sum_{i=1}^{n} d_i)+d_n} \leq a$, which is equivalent to the statement.

□

B.1.2 Assumptions on the signature

As mentioned in Section 4.4, functions of type $\tau_1 \multimap \tau_2$ are 1-sensitive in $\tau_1 \to \tau_2$. For the soundness of our type system, this invariant must be respected also by the types of the functions exported by the signature Σ. We formalize this property by introducing a notion of validity for signatures. We write $\Gamma \vdash_{\not<:} A : \tau$ to denote typing derivations that do not make use of the subtyping rule Sub. This is useful to derive a precise type τ for A on which to apply the metric.

Definition B.3 (Valid Signature)**.** *A signature Σ is valid if it only exports (core) type bindings of the form:*

1. *$c : b$ where $c : b$ is the only type definition for c in Σ*
2. *$f : \tau_1 \multimap \tau_2$ where f is 1-sensitive in the type $\tau_1 \to \tau_2$ and if $f\ F \xrightarrow{det}_1 C$ then $\emptyset \vdash_{\not<:} F : \tau_1$ and $\emptyset \vdash_{\not<:} C : \tau_2$.*

By encapsulating the reasoning about the sensitivity of deterministic functions within the signature, we make it possible to leverage existing results on function sensitivity and easily incorporate new primitives and datatypes in our framework. Notice that, for enhancing the expressivity of our type system, we allow the signature to export multiple types for the same function.

Example B.1. *Following the proof by Reed and Pierce [36], it can be shown that the addition operator $(+)$ is 1-sensitive in type $\phi = (!_1\mathbb{Z} \otimes !_1\mathbb{Z}) \multimap !_1\mathbb{Z}$ and $+ : \phi$ can thus be added to the signature.*

B.1.3 Distance of expressions

One of the fundamental properties ensured by our type system is *type and distance preservation*, that is, the type of and the distance between expressions is preserved by reduction. First, we introduce the notion of distance for expressions. This is defined on expressions that share the same structure and only differ in some of their constants. [1] Intuitively, the distance between similar expressions is defined by summing the distance between the differing constants.

[1] A similar structural property for expressions is used in ProVerif [20] and in the type system for Computational RCF [34] to deal with observational equivalence relations.

Definition B.4 (Constant Substitutions). *Let σ_1, σ_2 be substitutions mapping variables into constants such that $dom(\sigma_1) = dom(\sigma_2)$.*

We say that σ_1, σ_2 is a pair of constant substitutions *if for all $x \in dom(\sigma_1)$ there exists a core type ϕ such that $x\sigma_1 : \phi \in \Sigma$ and $x\sigma_2 : \phi \in \Sigma$.*

Definition B.5 (Similar Expressions). *We say that two closed expressions A_1 and A_2 are (A, σ_1, σ_2)-similar denoted by $A_1 \Bumpeq_{A;\sigma_1;\sigma_2} A_2$ for expression A and constant substitutions σ_1, σ_2 if $A_i = A\sigma_i$ for all $i \in \{1, 2\}$. We say A is the* skeleton *of A_1, A_2 with respect to σ_1, σ_2.*

We aim at showing that the distance between expressions is preserved by reduction: this holds true only if we take the typing environment that is used to type-check the corresponding skeleton into account. For instance, consider the skeleton expression $A := \mathsf{let}\ z = (y+y)\ \mathsf{in}\ z$ and the two substitutions $\sigma_1 := \{4/y\}$ and $\sigma_2 := \{5/y\}$. We have $A\sigma_1 \to^* 8$ and $A\sigma_2 \to^* 10$. The skeleton expression A can only be typed in a typing environment giving y type $!_2\mathbb{Z}$ (or a subtype thereof). If we define the distance between $A\sigma_1$ and $A\sigma_2$ as the distance between the corresponding constant substitutions multiplied by the replication index in the typing environment (that is, $1 \cdot 2$), we see that the distance is preserved by reduction. The intuition is that the typing environment used to type-check the skeleton expression statically characterizes how much the distance between the differing values may be amplified at run-time. For this reason, the notion of distance is parameterized by the typing environment used to type-check the skeleton expression.

Definition B.6 (Constant Environment). *Let σ_1, σ_2 be a pair of constant substitutions and Γ be a typing environment such that $\Gamma \vdash \diamond$ and $dom(\sigma_1) = dom(\Gamma)$.*

We call Γ a constant environment for σ_1, σ_2 if for all $x \in dom(\Gamma)$, $x\sigma_1 : \phi \in \Sigma$ implies $x :!_k\phi \in \Gamma$ for some k.

We can now define the notion of distance for pairs of constant substitutions and for expressions.

Definition B.7 (Distance of Constant Substitutions). *Let σ_1, σ_2 be a pair of constant substitutions and let Γ be a constant environment for σ_1, σ_2. We define the distance between σ_1 and σ_2 with respect to Γ as follows:*

$$\mathsf{dist}_\Gamma(\sigma_1, \sigma_2) = \sum_{x:\tau \in \Gamma} \delta_\tau(x\sigma_1, x\sigma_2).$$

Definition B.8 (Distance of Expressions). *Let A_1, A_2 be (A, σ_1, σ_2)-similar and Γ be a constant environment for σ_1, σ_2. The distance between A_1 and A_2 with respect to Γ is defined as $\mathsf{dist}_\Gamma(A_1, A_2) = \mathsf{dist}_\Gamma(\sigma_1, \sigma_2)$.*

These definitions similarly extend to stores and configurations.

B.1.4 Type and distance preservation

We first introduce two convenient notations. We define the notion of well-typedness for configurations, written $\Gamma \vdash [S, A] : \tau$. The idea is to split Γ so that the resulting environments can be used to give A type τ and to also type the content of each reference in S.

Definition B.9 (Well-typed Configurations)**.** *For all expressions A, stores $S = \{a_1 : \tau_1 \mapsto M_1, \ldots, a_n : \tau_n \mapsto M_n\}$, types τ, and environments Γ, we write that $\Gamma \vdash [S, A] : \tau$ if and only if there exist some environments $\Gamma_1, \ldots, \Gamma_n, \Gamma_A$ such that $\Gamma = \Gamma_1 + \ldots + \Gamma_n + \Gamma_A$ and $\Gamma_A \vdash A : \tau$ and $\Gamma_i \vdash M_i :!_1\mathsf{Option}\langle\tau_i\rangle$ for all $i \in [1, n]$. We refer to Γ_A as $env_A(\Gamma \vdash [S, A] : \tau)$.*

The second notation is used to characterize a set of possible continuations for a given expression.

Definition B.10 (Set of Continuations)**.** *Let $[S, A]$ be a configuration, $\widetilde{\ell} := \ell_1, \ldots, \ell_n$, $\widetilde{p} := p_1, \ldots, p_n$ and $\widetilde{[S', B]} := [S'_1, B_1], \ldots, [S'_n, B_n]$. We write $[S, A] \xrightarrow{\widetilde{\ell}}_{\widetilde{p}} \widetilde{[S', B]}$ if and only if $[S, A] \xrightarrow{\ell_i}_{p_i} [S'_i, B_i]$ for all $i \in [1, n]$. (Notice that there can be more than one continuation only if the primitive to be executed is $\mathsf{add\_noise}^s_{\mathbb{Z}\to\mathbb{Z}}\ M$ or $\mathsf{san\_}X^s_{b_1\to b_2}\ M$.)*

Proposition B.3 (Same Derivation for Set of Continuations)**.** *For all configurations $[S, A]$, sets of configurations $\widetilde{[S', B]}$, sets of probabilities $\widetilde{p}$, and sets of rules $\widetilde{\ell}$ it holds that if $[S, A] \xrightarrow{\widetilde{\ell}}_{\widetilde{p}} \widetilde{[S', B]}$ then all derivations of $[S, A] \xrightarrow{\ell_i}_{p_i} [S'_i, B_i]$ use the same semantic reduction rules.*

Definition B.11 (Structural Typing)**.** *We write $\Sigma \vdash A : \tau$ to denote that $\emptyset \vdash_{\not<:} A : \tau$.*

Lemma B.20 (Similar Functional Terms)**.** *Let F_1, F_2 be two functional terms, such that (M, σ_1, σ_2)-similar for some term M and some constant substitutions σ_1, σ_2. Let Γ be a constant environment for σ_1, σ_2. If $\Sigma \vdash F_i : \tau$ for all $j \in [1, 2]$ and some type τ then $\delta_\tau(F_1, F_2) \leq \mathsf{dist}_\Gamma(F_1, F_2)$.*

Proof. By induction on the structure of F_1, F_2. □

Lemma B.21 (Similar Constant Terms)**.** *For some constant terms C_1, C_2 such that $\Sigma \vdash C_j : \tau$ for all $j \in [1, 2]$ and some type τ it holds that if $\delta_\tau(C_1, C_2) < \infty$ then there exist two constant substitutions σ_1, σ_2, a term M, and a constant environment Γ for σ_1, σ_2 such that C_1, C_2 are (M, σ_1, σ_2)-similar and $\delta_\tau(C_1, C_2) = \mathsf{dist}_\Gamma(C_1, C_2)$.*

Proof. By induction on the structure of C_1, C_2. Since $\delta_\tau(C_1, C_2) < \infty$ it is easy to see that C_1, C_2 must share the same structure. The base case c_1, c_2 follows immediately by choosing $M := x$, $\sigma_i := \{c_i/x\}$, $\Gamma = x : \tau$ for some fresh variable x. It is easy to see, that these values fulfill the requirements. In the other cases we conclude using the induction hypothesis either once or twice. □

$[S, (\lambda x.A)N] \xrightarrow{\mathsf{det}}_1 [S, A\{N/x\}]$	Red-Fun
$[S, \mathsf{let}\ (x, y) = (M, N)\ \mathsf{in}\ ]A \xrightarrow{\mathsf{det}}_1 [S, A\{M/x\}\{N/y\}]$	Red-Split
$[S, \mathsf{case\ inl}\ M\ \mathsf{of}\ x\ \mathsf{in}\ A\ \mathsf{else}\ B\ ] \xrightarrow{\mathsf{det}}_1 [S, A\{M/x\}]$	Red-CaseL
$[S, \mathsf{case\ inr}\ M\ \mathsf{of}\ x\ \mathsf{in}\ A\ \mathsf{else}\ B\ ] \xrightarrow{\mathsf{det}}_1 [S, B\{M/x\}]$	Red-CaseR
$[S, \mathsf{unfold\ fold}\ M\ \mathsf{as}\ x\ \mathsf{in}\ A\ \mathsf{else}\ B\ ] \xrightarrow{\mathsf{det}}_1 [S, A\{M/x\}]$	Red-Match
$[S, \mathsf{unfold}\ M\ \mathsf{as}\ x\ \mathsf{in}\ A\ \mathsf{else}\ B\ ] \xrightarrow{\mathsf{det}}_1 [S, B] \quad \text{if } \forall N.M \neq \mathsf{fold}\ N$	Red-Match-Fail
$[S, \mathsf{let}\ x = M\ \mathsf{in}\ A] \xrightarrow{\mathsf{det}}_1 [S, A\{M/x\}]$	Red-LetVal
$[S, \mathsf{let}\ x = A\ \mathsf{in}\ B] \xrightarrow{\ell}_p [S', \mathsf{let}\ x = A'\ \mathsf{in}\ B] \quad \text{if } [S, A] \xrightarrow{\ell}_p [S', A']$	Red-Let
$[S, \mathsf{ref}_\tau] \xrightarrow{\mathsf{det}}_1 [S \uplus \{a : \tau \mapsto \mathsf{none}\}, (\mathsf{read}_{a:\tau}, \mathsf{write}_{a:\tau})]$	Red-Ref
$[S \cup \{a : \tau \mapsto M\}, \mathsf{read}_{a:\tau}()] \xrightarrow{\mathsf{det}}_1 [S \cup \{a : \tau \mapsto \mathsf{none}\}, M]$	Red-Read
$[S \cup \{a : \tau \mapsto M\}, \mathsf{write}_{a:\tau} N] \xrightarrow{\mathsf{det}}_1 [S \cup \{a : \tau \mapsto \mathsf{some}\ N\}, ()]$	Red-Write
$[S, \mathsf{add\_noise}^s_{\mathbb{Z}\to\mathbb{Z}}\ z_1] \xrightarrow{\textsc{Noise}\ (z_1,z_2,s)}_{\mathsf{DLap}^s(z_2)} [S, z]$ where $z_1, z_2 \in \mathbb{Z}$ and $z = z_1 +_{\mathbb{Z}} z_2$	Red-Noise
$[S, \mathsf{san\_}X^s_{b_1\to b_2}\ c_1] \xrightarrow{\textsc{San}\ (X^s_{b_1\to b_2}, c_1, c_2)}_p [S, c_2]$ where $p = Pr[X^s_{b_1\to b_2}(c_1) = c_2]$ and $c_i : b_i \in \Sigma$ and $X^s_{b_1\to b_2} \in \mathcal{X}$.	Red-San
$[S, M = M] \xrightarrow{\mathsf{det}}_1 [S, \mathsf{true}]$	Red-Eq-True
$[S, M = N] \xrightarrow{\mathsf{det}}_1 [S, \mathsf{false}] \quad \text{where } M \neq N$	Red-Eq-False
$[S, f\ F] \xrightarrow{\mathsf{det}}_1 [S, C] \quad \text{if } f(F) =_\Sigma C$	Red-Sig

Notation: We let $\mathcal{X}$ denote the set of all mechanisms included into the system via the general mechanism primitive.

Table B.1: Full semantics of extended RCF_{DF7}

The semantic rules of our calculus together with the reduction rule names are stated again in Table B.1, now also including the semantic rule for the general mechanism primitive $\mathsf{san\_}X^s_{b_1\to b_2}\ c_1$.

Note that our typing rules are not deterministic due to subtyping and rule $!_k$I of the form

$$\frac{\Gamma \vdash M : \tau}{k\Gamma \vdash M :!_k\tau}.$$

We assume that in the following proof we always consider the shortest typing derivation where all occurrences of

$$\cfrac{\cfrac{\Gamma \vdash M : \tau}{k'\Gamma \vdash M :!_{k'}\tau}}{k \cdot (k'\Gamma) \vdash M :!_k(!_{k'}\tau)}$$

are (recursively) replaced by the equivalent rule application

$$\frac{\Gamma \vdash M : \tau}{(k \cdot k')\Gamma \vdash M :!_{k\cdot k'}\tau}.$$

We furthermore assume that for all applications of $!_k$I $k \neq 1$ (since applying the rule for $k = 1$ is redundant). We also consider the shortest derivation w.r.t. subsequent applications of the subtyping rule. For instance,

$$\dfrac{\dfrac{\Gamma \vdash A : \tau \qquad \Gamma \vdash \tau <: \tau'}{\Gamma \vdash A : \tau'} \qquad \Gamma \vdash \tau' <: \tau''}{\Gamma \vdash A : \tau''}$$

can be (recursively) replaced with the equivalent derivation

$$\dfrac{\Gamma \vdash A : \tau \qquad \Gamma \vdash \tau <: \tau''}{\Gamma \vdash A : \tau''}$$

by using the transitivity of the subtyping relation (Lemma B.17).

We can now state the type and distance preservation theorem. Given two similar configurations, this theorem says that each reduction of the first configuration can be associated with a reduction of the second configuration (item 1) so that the target configurations are similar (item 2) and well-typed (item 3). In the case of det reductions (item 4a), the distance between the target configurations is smaller or equal to the one between the initial configurations and there exists exactly one possible reduction. In the case of noise addition (item 4b), the distance between the target configurations is the distance between the initial configurations minus the distance between the arguments of the $\mathsf{add\_noise}^{e^{-\epsilon/k}}_{\mathbb{Z}\to\mathbb{Z}}$ primitive (the noise in the two configurations is chosen so that $\mathsf{add\_noise}^{e^{-\epsilon/k}}_{\mathbb{Z}\to\mathbb{Z}}\ z_1$ and $\mathsf{add\_noise}^{e^{-\epsilon/k}}_{\mathbb{Z}\to\mathbb{Z}}\ z_2$ give the same result) and, for each set of corresponding reductions, the ratio between the reduction probabilities is bounded by $e^{\frac{\epsilon\cdot\delta_{\mathbb{Z}}(z_1,z_2)}{k}}$. This bound plays a crucial role in linking distance preservation and differential privacy. The case of the general mechanism primitive (item 4c) closely resembles the previous case. In this case, the distance between the target configurations is the distance between the initial configurations minus the distance between the arguments c_1, c_2 of the $\mathsf{san\_}X^s_{b\to b'}$ primitive. The ratio between the probabilities of the mechanism $X^s_{b\to b'}$ reducing to the same result $c'_1 = c'_2$ in both reductions is bounded by $e^{\frac{\epsilon\cdot\delta_b(c_1,c_2)}{k}}$, which is crucial in linking distance preservation and differential privacy. If no additional mechanism is included in the calculus via the general mechanism primitive (i.e., $\mathcal{X} = \emptyset$), this item can be dropped.

Theorem B.1 (Type and Distance Preservation). *Let $[S_1, A_1]$ and $[S_2, A_2]$ be $([S, A], \sigma_1, \sigma_2)$-similar. Let τ be a type, Γ a typing environment, and Γ' a constant environment for σ_1, σ_2, where* $\mathsf{dist}_{\Gamma'}([S_1, A_1], [S_2, A_2]) < \infty$.

Let $\mathcal{X}$ denote the set of all noise mechanisms that are included into the system via the general mechanism primitive occurring in $[S, A]$. We assume the following conditions to hold:

- *the parameter of all noise addition primitives occurring in $[S, A]$ is set to $s := k/\epsilon$ (i.e., they are of the form* $\mathsf{add\_noise}^{k/\epsilon}_{\mathbb{Z}\to\mathbb{Z}}\ M$*);*

- *the parameter of all general mechanism primitives* $\mathsf{san\_}X^s_{b\to b'}\ M$ *for the mechanism* $X^s_{b\to b'} \in \mathcal{X}$ *occurring in* $[S, A]$ *is set to* $s := \epsilon/k$ *(i.e., they are of the form* $\mathsf{san\_}X^{\epsilon/k}_{b\to b'}\ M$*) and the respective mechanism* $X^s_{b\to b'}$ *provides* s*-differential privacy.*

If $\Gamma, \Gamma' \vdash [S, A] : \tau$ *and* $[S_1, A_1] \xrightarrow{\widetilde{\ell_1}}_{\widetilde{p_1}} \widetilde{[T_1, B_1]}$ *then there exist a set of labels* $\widetilde{\ell_2}$ *(with* $\left|\widetilde{\ell_1}\right| = \left|\widetilde{\ell_2}\right|$*), a set of configurations* $\widetilde{[T_2, B_2]}$*, a set of probabilities* $\widetilde{p_2}$*, and an environment* Γ'' *such that*

1. $[S_2, A_2] \xrightarrow{\widetilde{\ell_2}}_{\widetilde{p_2}} \widetilde{[T_2, B_2]}$*;*

Furthermore, for every $(\ell_2, p_2, [T_2, B_2]) \in \widetilde{(\ell_2, p_2, [T_2, B_2])}$*, there exists a distinct* $(\ell_1, p_1, [T_1, B_1]) \in \widetilde{(\ell_1, p_1, [T_1, B_1])}$*, a skeleton* $[T, B]$*, and a pair of constant substitutions* σ'_1, σ'_2 *for which* Γ'' *is a constant environment such that*

2. $[T_1, B_1] \bumpeq_{[T,B];\sigma'_1;\sigma'_2} [T_2, B_2]$*,*

3. $\Gamma, \Gamma'' \vdash [T, B] : \tau$*, and*

4. *the distance and reduction probabilities are bounded as follows:*

   (a) *either* $p_1 = p_2 = 1$ *and* $\mathsf{dist}_{\Gamma''}([T_1, B_1], [T_2, B_2]) \leq \mathsf{dist}_{\Gamma'}([S_1, A_1], [S_2, A_2])$

   (b) *or* $\ell_i = \textsc{Noise}\ (z_i, z'_i, k/\epsilon)$ *for* $i \in \{1, 2\}$ *and* $z_1 + z'_1 = z_2 + z'_2$ *and* $\mathsf{dist}_{\Gamma''}([T_1, B_1], [T_2, B_2]) = \mathsf{dist}_{\Gamma'}([S_1, A_1], [S_2, A_2]) - \delta_{\mathbb{Z}}(z_1, z_2)$ *and* $\frac{p_1}{p_2} \leq e^{\frac{\epsilon \cdot \delta_{\mathbb{Z}}(z_1, z_2)}{k}}$

   (c) *or* $\ell_i = \textsc{San}\ (X^{\epsilon/k}_T, c_i, c'_i)$ *for* $i \in \{1, 2\}$ *and* $c'_1 = c'_2$ *and* $\mathsf{dist}_{\Gamma''}([T_1, B_1], [T_2, B_2]) = \mathsf{dist}_{\Gamma'}([S_1, A_1], [S_2, A_2]) - \delta_b(c_1, c_2)$ *and* $\frac{p_1}{p_2} \leq e^{\frac{\epsilon \delta_b(c_1, c_2)}{k}}$*, where* $T = b \to b'$ *and* $X^{\epsilon/k}_T \in \mathcal{X}$*.*

Proof. Notation: By Definition B.9 we know that the store S must have length $n \in \mathbb{N}$ and there must exist environments $\Gamma_1, \Gamma'_1 \ldots, \Gamma_n, \Gamma'_n, \Gamma_A, \Gamma'_A$ such that $\Gamma = \Gamma_1 + \ldots + \Gamma_n + \Gamma_A$ and $\Gamma' = \Gamma'_1 + \ldots + \Gamma'_n + \Gamma'_A$, where $\Gamma_A, \Gamma'_A \vdash A : \tau$ and Γ_i, Γ'_i types the i-th entry of S for all $i \in [1, n]$. We will denote $\Gamma_{\mathcal{C}} := \Gamma_1 + \ldots + \Gamma_n$ and $\Gamma'_{\mathcal{C}} := \Gamma'_1 + \ldots + \Gamma'_n$.

By Proposition B.3 we know that all derivations in $[S_1, A_1] \xrightarrow{\widetilde{\ell_1}}_{\widetilde{p_1}} \widetilde{[T_1, B_1]}$ have the same structure and depth. The proof is by induction on the depth of the derivations in $[S_1, A_1] \xrightarrow{\widetilde{\ell_1}}_{\widetilde{p_1}} \widetilde{[T_1, B_1]}$. We first consider all rules but Red-Sig, Red-Noise, Red-Exp, Red-San, and Red-Let. These rules are all deterministic, meaning there exists only one continuation $[T_1, B_1] := \widetilde{[T_1, B_1]}$ and $p_1 := \widetilde{p_1} = 1$ and they are all labeled with $l_1 := \widetilde{l_1} = \mathsf{det}$.

By inspection of the rules and basic properties of substitution it immediately follows that in all of these cases there exists a configuration $[T_2, B_2]$ such that $[S_2, A_2] \xrightarrow{\mathsf{det}}_1 [T_2, B_2]$, where $[T_1, B_1] \bumpeq_{[T,B];\sigma_1;\sigma_2} [T_2, B_2]$, for some skeleton $[T, B]$.

We will show this exemplarily for the rule Red-Split and omit it in the other cases.

Furthermore, we will also show that $\Gamma, \Gamma' \vdash [T, B] : \tau$. It then immediately follows that (item 4a) is fulfilled since $\mathsf{dist}_{\Gamma'}([T_1, B_1], [T_2, B_2]) = \mathsf{dist}_{\Gamma'}(\sigma_1, \sigma_2) = \mathsf{dist}_{\Gamma'}([S_1, A_1], [S_2, A_2])$.

Case Red-Split: We know that A_1 must be of the form $A_1 = \mathsf{let}\ (x, y) = (M_1, N_1)\ \mathsf{in}\ P_1$. This can be rewritten as $A_1 = \mathsf{let}\ (x, y) = (M\sigma_1, N\sigma_1)\ \mathsf{in}\ P\sigma_1$ and as $A_1 = (\mathsf{let}\ (x, y) = (M, N)\ \mathsf{in}\ P)\sigma_1$, where A must be of the form $A = \mathsf{let}\ (x, y) = (M, N)\ \mathsf{in}\ P$.

This also implies that there exists a B_1 of the form $B_1 = P_1\{M_1/x\}\{N_1/y\}$ which is rewritable as $B_1 = P\sigma_1\{M\sigma_1/x\}\{N\sigma_1/y\}$ and $B_1 = (P\{M/x\}\{N/y\})\sigma_1$.

We additionally know that since A_1 and A_2 share the same structure it must be the case that $A_2 = (\mathsf{let}\ (x, y) = (M, N)\ \mathsf{in}\ P)\sigma_2 = \mathsf{let}\ (x, y) = (M\sigma_2, N\sigma_2)\ \mathsf{in}\ P\sigma_2$ $= \mathsf{let}\ (x, y) = (M_2, N_2)\ \mathsf{in}\ P_2$ and that thus by the definition of Red-Split it holds that $[S_2, A_2] \xrightarrow{\mathsf{det}}_1 [S_2, B_2]$ where

$$B_2 = P_2\{M_2/x\}\{N_2/y\} = P\sigma_2\{M\sigma_2/x\}\{N\sigma_2/y\} = (P\{M/x\}\{N/y\})\sigma_2.$$

Thus, we can immediately see that $[S_1, B_1] \mathrel{\tilde{\circ}}_{[S,B];\sigma_1;\sigma_2} [S_2, B_2]$ where $B = P\{M/x\}\{N/y\}$.

Since $\Gamma_A, \Gamma'_A \vdash A : \tau$ we know by Lemma B.18 and the definition of the only applicable typing rule $\otimes$E that there must exist r, Δ_1, Δ_2 such that $\Gamma_A, \Gamma'_A = r\Delta_1 + \Delta_2$, where $\Delta_1 \vdash (M, N) :!_1(\tau_1 \otimes \tau_2)$ for some τ_1, τ_2 and $\Delta_2, x :_r \tau_1, y :_r \tau_2 \vdash P : \rho$ for some type ρ such that $\Gamma_A, \Gamma'_A \vdash \rho <: \tau$.

By Lemma B.18 and the definition of $\otimes$I it must furthermore be the case that there exist Δ_1^M, Δ_1^N such that $\Delta_1 = \Delta_1^M + \Delta_1^N$ and that $\Delta_1^M \vdash M : \tau_1'$ and $\Delta_1^N \vdash N : \tau_2'$ for some types τ_1', τ_2' such that $\Delta_1^M + \Delta_1^N \vdash !_1(\tau_1' \otimes \tau_2') <: !_1(\tau_1 \otimes \tau_2)$.

By Lemma B.18 this implies that the last applied rule in the subtyping derivation must have been Sub Pair and thus $\Delta_1^M \vdash \tau_1' <: \tau_1$ and $\Delta_1^N \vdash \tau_2' <: \tau_2$. We apply Sub to derive that $\Delta_1^M \vdash M : \tau_1$ and $\Delta_1^N \vdash N : \tau_2$.

By applying Lemma B.12 twice we can derive that

$$r\Delta_1^M + r\Delta_1^N + \Delta_2 \vdash P\{M/x\}\{N/y\} : \rho.$$

We apply rule Sub and derive that $r\Delta_1^M + r\Delta_1^N + \Delta_2 \vdash P\{M/x\}\{N/y\} : \tau$, which is equivalent to $\Gamma_A, \Gamma'_A \vdash P\{M/x\}\{N/y\} : \tau$. Since $S = T$ by Definition B.9 we can conclude that $\Gamma, \Gamma' \vdash [T, B] : \tau$.

Case Red-Fun: In this case we can apply the same reasoning as above to derive that the skeleton A must be of the form $A = (\lambda x.P)N$ and that there exists a skeleton $B = P\{N/x\}$ such that for $T := S$ the skeleton configuration $[T, B]$ fulfills all necessary criteria. It remains to be shown that $\Gamma, \Gamma' \vdash [T, B] : \tau$.

By multiple applications of Lemma B.18 and the definitions of $\multimap$E and $\multimap$I it must be the case that

$\Gamma_A, \Gamma'_A = \Delta_1 + \Delta_2$ for some environments Δ_1, Δ_2 such that

- $\Delta_1 \vdash \lambda x.P :!_1(\tau_1 \multimap \tau_2)$,
- $\Delta_2 \vdash N : \tau_1$,
- $\Gamma_A, \Gamma'_A \vdash \tau_2 <: \tau$,
- $\Delta_1, x :_1 \tau'_1 \vdash P : \tau'_2$,
- $\Delta_1 \vdash !_1(\tau'_1 \multimap \tau'_2) <:!_1(\tau_1 \multimap \tau_2)$,
- $\Delta_1 \vdash \tau_1 <: \tau'_1$,
- $\Delta_1 \vdash \tau'_2 <: \tau_2$

for some types $\tau_1, \tau_2, \tau'_1, \tau'_2$.

We can apply the rule SUB to derive that $\Delta_2 \vdash N : \tau'_1$ and that $\Delta_1, x :_1 \tau'_1 \vdash P : \tau_2$.

By applying Lemma B.12 to these two premises we can derive that $1 \cdot \Delta_2 + \Delta_1 \vdash P\{N/x\} : \tau_2$.

We apply the rule SUB to derive that $\Gamma_A, \Gamma'_A \vdash P\{N/x\} : \tau$.

Since $S = T$ by Definition B.9 we can conclude that $\Gamma, \Gamma' \vdash [T, B] : \tau$.

Case RED-CASEL, RED-CASER, RED-MATCH, RED-MATCH-FAIL, and RED-LETVAL : These cases are standard and analogous to the previous two, repeatedly making use of rule SUB, Lemma B.18, and Lemma B.9, whenever necessary. We omit the details and focus on the more interesting cases.

Case RED-EQ-TRUE: We use the same reasoning as in previous cases to derive that the skeleton A must be of the form $A = (M = N)$ and that there exists a skeleton $B = \mathsf{true}$ such that for $T := S$ the skeleton configuration $[T, B]$ fulfills all necessary criteria (note that $B\sigma_1 = B\sigma_2 = B = \mathsf{true}$).

We note that $\mathsf{true} \triangleq \mathsf{inl}\ ()$ and $\mathsf{Bool} \triangleq !_\infty\mathsf{Unit}+!_\infty\mathsf{Unit}$.

It remains to be shown that $\Gamma, \Gamma' \vdash [T, B] : \tau$.

By Lemma B.18 and the definition of EQUAL it must be the case that $\Gamma_A, \Gamma'_A = \Delta_1 + \Delta_2$ for some environments Δ_1, Δ_2 such that

- $\Delta_1 \vdash M :!_\infty\rho$,
- $\Delta_2 \vdash N :!_\infty\rho$,
- $\Gamma_A, \Gamma'_A \vdash !_\infty\mathsf{Bool} <: \tau$,

for some type ρ.

By applying SIG we can derive that $\emptyset \vdash () :!_1\mathsf{Unit}$. We apply $!_k$I to derive that $\infty\emptyset \vdash () :!_\infty\mathsf{Unit}$. By $+/\mu$I it follows that $\infty\emptyset \vdash \mathsf{inl}\ () :!_1(!_\infty\mathsf{Unit}+!_\infty\mathsf{Unit})$ and thus by $!_k$I it holds that $\underbrace{\infty\infty\emptyset}_{\emptyset} \vdash \mathsf{inl}\ () : \underbrace{!_\infty(!_\infty\mathsf{Unit}+!_\infty\mathsf{Unit})}_{!_\infty\mathsf{Bool}}$.

We apply Lemma B.9 to derive that $\Gamma_A, \Gamma'_A \vdash \mathsf{true} :!_\infty\mathsf{Bool}$ and using SUB conclude that $\Gamma_A, \Gamma'_A \vdash \mathsf{true} : \tau$.

Since $S = T$ by Definition B.9 we can conclude that $\Gamma, \Gamma' \vdash [T, B] : \tau$.

Case Red-Eq-False: Analogous to the previous case Red-Eq-True.

Case Red-Ref: We use the same reasoning as in previous cases to derive that the skeleton A must be of the form $A = \mathsf{ref}_\rho$ for some type ρ and there exists a skeleton $B = (\mathsf{read}_{a:\rho}, \mathsf{write}_{a:\rho})$ such that for $T := S \uplus \{a : \rho \mapsto \mathsf{none}\}$ the skeleton configuration $[T, B]$ fulfills all necessary criteria.

Note that there are two typing rules for references: Ref and Ref Opp. We assume without loss of generality that the references are typed using Ref. The proof for Ref Opp is analogous and only changes minimally by replacing Read by Read Opp in the following. By Lemma B.18 and the definition of Ref it must be the case that $\Gamma_A, \Gamma'_A \vdash \mathsf{ref}_\rho :!_\infty\mathsf{Ref}\langle\rho\rangle$ and $\Gamma_A, \Gamma'_A \vdash !_\infty\mathsf{Ref}\langle\rho\rangle <: \tau$.

We note that $\mathsf{Ref}\langle\rho\rangle \triangleq !_\infty\mathsf{Read}\langle\rho\rangle \otimes !_\infty\mathsf{Write}\langle\rho\rangle$.

Let Δ_∞ be an exponential fragment of Γ_A, Γ'_A.

Using Read and $!_k$I we derive that $\Delta_\infty \vdash \mathsf{read}_{a:\rho} :!_\infty\mathsf{Read}\langle\rho\rangle$.

Using Write and $!_k$I we derive that $\Delta_\infty \vdash \mathsf{write}_{a:\rho} :!_\infty\mathsf{Write}\langle\rho\rangle$.

By application of $\otimes$I and $!_k$I we derive that

$$\Delta_\infty \vdash (\mathsf{read}_{a:\rho}, \mathsf{write}_{a:\rho}) :!_\infty(!_\infty\mathsf{Read}\langle\rho\rangle \otimes !_\infty\mathsf{Write}\langle\rho\rangle).$$

By Lemma B.9 and Lemma B.7 it follows that $\Gamma_A, \Gamma'_A \vdash B :!_\infty\mathsf{Ref}\langle\rho\rangle$. Using Sub we deduce that $\Gamma_A, \Gamma'_A \vdash B :!_\infty\tau$.

Additionally, we can easily show that $\emptyset \vdash \mathsf{none} :!_1\mathsf{Option}\langle\rho\rangle$, by unfolding the syntactic definitions of none and $\mathsf{Option}\langle\rho\rangle$ (similar to the typing of true in case Red-Eq-True). This allows us to conclude that $\Gamma, \Gamma' \vdash [T, B] : \tau$ by Definition B.9.

Case Red-Read: We use reasoning similar to the previous cases to derive that there exists a $j \in [1, n]$ such that the skeleton store S must be of the form $S = S' \cup \{a_j : \tau_j \mapsto M_j\}$ for some skeleton store S'. and that the skeleton expression A must be of the form $A = \mathsf{read}_{a_j:\tau_j}()$. Furthermore, there exists a skeleton B of the form $B = M_j$, and skeleton store $T = S \cup \{a_j : \tau_j \mapsto \mathsf{none}\}$.

Note that as in the previous case there are two typing rules for reference reading: Read and Read Opp. We assume without loss of generality that the references are typed using Read, the other case follows very similarly.

By Lemma B.18 and the definitions of $\multimap$E and Read it must be the case that $\Gamma_A, \Gamma'_A = \Delta_1 + \Delta_2$ for some environments Δ_1, Δ_2 such that

- $\Delta_1 \vdash \mathsf{read}_{a_j:\tau_j} :!_1(\rho_1 \multimap \rho_2)$,
- $\Delta_2 \vdash () : \rho_1$,
- $\Gamma_A, \Gamma'_A \vdash \rho_2 <: \tau$,
- $\Delta_1 \vdash \mathsf{read}_{a_j:\tau_j} :!_1(!_\infty\mathsf{Unit} \multimap !_1\mathsf{Option}\langle\tau_j\rangle)$,
- $\Delta_1 \vdash !_1(!_\infty\mathsf{Unit} \multimap !_1\mathsf{Option}\langle\tau_j\rangle) <:!_1(\rho_1 \multimap \rho_2)$,

- $\Delta_1 \vdash \rho_1 <:!_\infty \mathsf{Unit}$,
- $\Delta_1 \vdash \mathsf{Option}\langle\tau_j\rangle <: \rho_2$

for some types ρ_1, ρ_2.

We know by Definition B.9 that $\Gamma_j, \Gamma'_j \vdash M_j : \mathsf{Option}\langle\tau_j\rangle$.

We can apply Lemma B.9 to derive that $\Gamma_A, \Gamma'_A \vdash \mathsf{Option}\langle\tau_j\rangle <: \rho_2$. This allows us to apply Lemma B.17 to yield $\Gamma_A, \Gamma'_A \vdash \mathsf{Option}\langle\tau_j\rangle <: \tau$. Since both Γ_A, Γ'_A and Γ_j, Γ'_j are fragments of Γ, Γ' we can use Lemma B.9 and Lemma B.7 to derive that $\Gamma_j, \Gamma'_j \vdash \mathsf{Option}\langle\tau_j\rangle <: \tau$.

Thus, an application of SUB results in $\Gamma_j, \Gamma'_j \vdash M_j : \tau$.

It remains to be shown that $\Gamma_A, \Gamma'_A \vdash \mathsf{none} : \mathsf{Option}\langle\tau_j\rangle$. This follows similarly to the previous case RED-REF, by first proving the easier statement $\Delta_\infty \vdash \mathsf{none} : \mathsf{Option}\langle\tau_j\rangle$ for some exponential fragment Δ_∞ of Γ_A, Γ'_A and then applying Lemma B.9.

This allows us to conclude that $\Gamma, \Gamma' \vdash [T, B] : \tau$ by Definition B.9.

Case RED-WRITE: Similar to the previous case we derive that there exists a $j \in [1, n]$ such that the skeleton store S must be of the form $S = S' \cup \{a_j : \tau_j \mapsto M_j\}$ for some skeleton store S'. and that the skeleton expression A must be of the form $A = \mathsf{write}_{a_j:\tau_j} N$. Furthermore, there exists a skeleton B of the form $B = ()$, and skeleton store $T = S \cup \{a_j : \tau_j \mapsto \mathsf{some}\ N\}$.

By Lemma B.18 and the definitions of $\multimap$E and WRITE it must be the case that $\Gamma_A, \Gamma'_A = \Delta_1 + \Delta_2$ for some environments Δ_1, Δ_2 such that

- $\Delta_1 \vdash \mathsf{write}_{a_j:\tau_j} :!_1(\rho_1 \multimap \rho_2)$,
- $\Delta_2 \vdash N : \rho_1$,
- $\Gamma_A, \Gamma'_A \vdash \rho_2 <: \tau$,
- $\Delta_1 \vdash \mathsf{write}_{a_j:\tau_j} :!_1(\tau_j \multimap !_\infty \mathsf{Unit})$,
- $\Delta_1 \vdash !_1(\tau_j \multimap !_\infty \mathsf{Unit}) <:!_1(\rho_1 \multimap \rho_2)$,
- $\Delta_1 \vdash \rho_1 <: \tau_j$,
- $\Delta_1 \vdash !_\infty \mathsf{Unit} <: \rho_2$

for some types ρ_1, ρ_2.

Using Lemma B.9 it follows that $\Gamma_A, \Gamma'_A \vdash N : \rho_1$ and that $\Gamma_A, \Gamma'_A \vdash \rho_1 <: \tau_j$.

We apply SUB to derive that $\Gamma_A, \Gamma'_A \vdash N : \tau_j$.

We know that $\mathsf{some}\ B \triangleq \mathsf{inr}\ N$ and that $\mathsf{Option}\langle\tau_j\rangle =!_\infty \mathsf{Unit} + \tau_j$. Using $+/\mu$ I we derive that $\Gamma_A, \Gamma'_A \vdash \mathsf{some}\ N : \mathsf{Option}\langle\tau_j\rangle$

It can be easily seen that as in previous cases by applying SIG and $!_k$I together with Lemma B.9 and Lemma B.7 and Lemma B.17 we can derive that $\Gamma_j, \Gamma'_j \vdash () :!_\infty \mathsf{Unit} <: \tau$.

This allows us to conclude that $\Gamma, \Gamma' \vdash [T, B] : \tau$ by Definition B.9.

We now consider the next deterministic case Red-Sig.

Case Red-Sig: Again, there exists only one continuation $[T_1, B_1] := \widetilde{[T_1, B_1]}$ and $p_1 := \widetilde{p_1} = 1$ and the reduction is labeled with $l_1 := \widetilde{l_1} = \mathsf{det}$.

We know that $[S_1, A_1]$ and $[S_2, A_2]$ are $([S, A], \sigma_1, \sigma_2)$-similar.

Thus, since the $dom(\sigma_i)$ for $i \in [1, 2]$ only contains constants A must be of the form $A = f\ M$ for some term M such that $M\sigma_j =: F_j$ and $A_j = f\ F_j$ for $j \in [1, 2]$.

Furthermore, $B_j = C_j$, where $f(F_j) =_\Sigma C_j$ for $j \in [1, 2]$.

Note that the stores S_1, S_2 are unaffected by the reduction rule, so $T_j := S_j$. For the sake of readability we will completely drop the stores from the following reasoning on distance preservation. They could be included in the reasoning by separating the substitutions σ_1, σ_2 into the parts needed for proving the structural equivalence of A_1, A_2 and the one needed for proving structural equivalence of S_1, S_2.

Since we know that $\Gamma, \Gamma' \vdash [S, f\ M] : \tau$ by Lemma B.18 we know that $f : \tau_1 \multimap \tau_2 \in \Sigma$ for some type τ_1, τ_2.

By Definition B.13 we know that this implies that $\Sigma \vdash F_i : \tau_1$ and $\Sigma \vdash C_i : \tau_2$.

This allows us to apply Lemma B.20 and to derive that $\delta_{\tau_1}(F_1, F_2) \leq \mathsf{dist}_{\Gamma'}(F_1, F_2)$. Notice that by Definition B.7 and Definition B.8 this also implies that $\delta_{\tau_1}(F_1, F_2) \leq \mathsf{dist}_{\Gamma'}(F_1, F_2) = \mathsf{dist}_{\Gamma'}(\sigma_1, \sigma_2) = \mathsf{dist}_{\Gamma'}(A_1, A_2)$.

Using Definition B.13 we also know that f is 1-sensitive in $\tau_1 \to \tau_2$, which by Definition 4.4 implies that $\delta_{\tau_2}(f(F_1), f(F_2)) \leq \delta_{\tau_1}(F_1, F_2)$, which is equivalent to $\delta_{\tau_2}(C_1, C_2) \leq \delta_{\tau_1}(F_1, F_2)$.

Adding this to the list of previous inequalities leads to $\delta_{\tau_2}(C_1, C_2) \leq \delta_{\tau_1}(F_1, F_2) \leq \mathsf{dist}_{\Gamma'}(F_1, F_2) = \mathsf{dist}_{\Gamma'}(\sigma_1, \sigma_2) = \mathsf{dist}_{\Gamma'}(A_1, A_2)$.

Since $\mathsf{dist}_{\Gamma'}([S_1, A_1], [S_2, A_2]) < \infty$ by assumption, we can apply Lemma B.21. Thus, there exist two constant substitutions σ'_1, σ'_2, a term M', and a constant environment Γ'' for σ'_1, σ'_2 such that C_1, C_2 are $(M', \sigma'_1, \sigma'_2)$-similar and $\delta_{\tau_2}(C_1, C_2) = \mathsf{dist}_{\Gamma''}(C_1, C_2)$.

It immediately follows that

$$\begin{aligned} & \mathsf{dist}_{\Gamma''}(C_1, C_2) = \delta_{\tau_2}(C_1, C_2) \leq \delta_{\tau_1}(F_1, F_2) \\ \leq\ & \mathsf{dist}_{\Gamma'}(F_1, F_2) = \mathsf{dist}_{\Gamma'}(\sigma_1, \sigma_2) = \mathsf{dist}_{\Gamma'}(A_1, A_2). \end{aligned}$$

The only remaining part (item 3) of the statement follows easily by Definition B.9 using the fact that $S = T$ as well as Lemma B.18 together with the definition of the rules $\multimap$E and Sig.

We now consider the two non-deterministic base cases Red-Noise and Red-San. Their proofs follow the same strategy..

Case Red-Noise: In this case it must be the case that $A_1 = \mathsf{add\_noise}^s_{\mathbb{Z}\to\mathbb{Z}}\ z_1$ for $s = e^{-\epsilon/k}$ and some $z_1 \in \mathbb{Z}$ and all $T_1 \in \widetilde{T_1}$ are equal to S_1.

It is easy to see that either $A_2 = A_1 = \mathsf{add\_noise}^s_{\mathbb{Z}\to\mathbb{Z}}\ z_1$ or $A_2 = \mathsf{add\_noise}^s_{\mathbb{Z}\to\mathbb{Z}}\ z_2$, for some $z_2 \in \mathbb{Z}$ such that $z_2 \neq z_1$.

In the first case, the first two items to be shown follow immediately. Since the proof of (item 3) and (item 4b) is equivalent to the one in the second case, we will solely focus on that one.

We can immediately deduce that $A = \mathsf{add\_noise}^s_{\mathbb{Z}\to\mathbb{Z}}\ x$, where $x\sigma_j = z_j$ for $j \in [1, 2]$.

Furthermore, any rule $\ell_{1i} \in \widetilde{\ell_1}$ must be of the form $\ell_{1i} = \text{Noise}\ (z_1, z'_{1i}, s)$ for some $z'_{1i} \in \mathbb{Z}$. Furthermore, this implies that $p_{1i} = \mathsf{DLap}^s(z'_{1i})$ and $T_{1i} = S_1$ and $B_{1i} = \underbrace{z_1 +_{\mathbb{Z}} z'_{1i}}_{=:z''_{1i}}$.

We choose $\ell_{2i} := \text{Noise}\ (z_2, z'_{2i}, s)$, where $z'_{2i} := z''_{1i} -_{\mathbb{R}} z_2$, $p_{2i} := \mathsf{DLap}^s(z'_{2i})$ and $T_{2i} := S_2$ and $B_{2i} := z_2 +_{\mathbb{Z}} z'_{2i}$. It follows immediately that $B_{2i} = z''_{1i} = B_{1i}$.

We can easily verify that this fulfills (item 1).

We now show that for every $(\ell_{2i}, p_{2i}, [T_{2i}, B_{2i}])$ for $i \in [1, n]$ the tuple

$$(\ell_{1i}, p_{1i}, [T_{1i}, B_{1i}])$$

satisfies the remaining conditions (item 2), (item 3), and (item 4b).

For the sake of readability, we now fix i for the remainder of this proof and drop it from all the previous indexes.

We know that $T_1 = S_1$ and $T_2 = S_2$. Thus we define $T := S$ as a skeleton store for T_1, T_2. Since $B_1 = B_2$ we choose $B := B_1$ as an appropriate skeleton expression for B_1, B_2.

We know that $z_1, z_2 \in \mathbb{Z}$ and that $x\sigma_j = z_j$ for $j \in [1, 2]$ and that thus $x :!_l\mathbb{Z} \in \Gamma'$ for some index l. We select $\Gamma'' := \Gamma'\{x :!_{l-1}\mathbb{Z}/x :!_l\mathbb{Z}\}$. Furthermore, we define the substitutions σ'_1, σ'_2 as follows: $y\sigma'_j := \begin{cases} y\sigma_j & \text{if } y \neq x \\ y\sigma_j & \text{if } y = x \text{ and } l > 1 \\ \bot & \text{otherwise} \end{cases}$

This means, that if $l = 1$ then x is neither contained in the domain of Γ'' nor in the domain of σ'_1, σ'_2. If $l > 1$ the substitutions and the domain of Γ' remain unchanged.

In the latter case, (item 2) follows immediately, since the substitutions have not changed.

In the former case, x is no longer contained in the domain of the constant environment and that of the constant substitutions. Since $B_1 = B_2$ it follows immediately that $B_1 \Bumpeq_{B;\sigma'_1;\sigma'_2} B_2$, but proving that $T_1 \Bumpeq_{T;\sigma'_1;\sigma'_2} T_2$ requires some more reasoning: we know that $x :!_1\mathbb{Z} \in \Gamma'$ and that $\Gamma, \Gamma' \vdash [S, \mathsf{add\_noise}^s_{\mathbb{Z}\to\mathbb{Z}}\ x] :$

τ. Since $x \in fv(\mathsf{add\_noise}^{s}_{\mathbb{Z}\to\mathbb{Z}}\ x)$, it must be the case that $x :!_1\mathbb{Z} \in \Gamma'_A$, since this binding is needed to type-check the skeleton expression $\mathsf{add\_noise}^{s}_{\mathbb{Z}\to\mathbb{Z}}\ x$. This in particular means that by the definition of environment addition $x \notin dom(\Gamma'_{\mathcal{C}})$ and thus $x \notin fv(S_j)$. This immediately implies that $S\sigma_j = S\sigma'_j$.

We now focus on (item 3).

We know that $\Gamma_A, \Gamma'_A \vdash \mathsf{add\_noise}^{s}_{\mathbb{Z}\to\mathbb{Z}}\ x : \tau$ and that $\Gamma' = \Gamma'_{\mathcal{C}} + \Gamma'_A$. As seen above it must be the case that $x \in dom(\Gamma'_A)$. We define $\Gamma''_{\mathcal{C}} := \Gamma'_{\mathcal{C}}$ and Γ''_A in such a way that $\Gamma'_A = \Gamma''_A + x :!_1\mathbb{Z}$.

By Lemma B.18 and rule Add-Noise it must be the case that $\Gamma_A, \Gamma'_A \vdash \mathsf{add\_noise}^{s}_{\mathbb{Z}\to\mathbb{Z}}\ x :!_\infty\mathbb{Z}$ and $\Gamma_A, \Gamma'_A \vdash !_\infty\mathbb{Z} <: \tau$.

By Lemma B.6 and Lemma B.8 it follows that $\Gamma_A, \Gamma''_A \vdash !_\infty\mathbb{Z} <: \tau$ (note that variable bindings have no influence on subtyping).

We know that $B = z''_1 \in \mathbb{Z}$. By rule Sig it holds that $\emptyset \vdash B :!_1\mathbb{Z}$. We apply $!_k$I to derive that $\emptyset \vdash B :!_\infty\mathbb{Z}$. Using Lemma B.9 this also implies that $\Gamma_A, \Gamma''_A \vdash B : !_\infty\mathbb{Z}$. Applying Sub yields $\Gamma_A, \Gamma''_A \vdash B : \tau$.

Since $T = S$ and $\Gamma_{\mathcal{C}}, \Gamma'_{\mathcal{C}} = \Gamma_{\mathcal{C}}, \Gamma''_{\mathcal{C}}$ by Definition B.9 it follows that $\Gamma, \Gamma'' \vdash [T, B] : \tau$.

The first part of (item 4b) follows immediately by our selection of $\Gamma'', \sigma'_1, \sigma'_2$, Definition B.7 and Definition B.8. The second part follows from the relation between noise addition, sensitivity, and differential privacy as shown by Chan et al. [116].

Case Red-San: In this case it must be the case that $A_1 = \mathsf{san}\_X^{s}_{b\to b'}\ c_1$ for $s = \epsilon/k$ and some $c_1 \in b$ and all $T_1 \in \widetilde{T_1}$ are equal to S_1.

It is easy to see that either $A_2 = A_1 = \mathsf{san}\_X^{s}_{b\to b'}\ c_1$ or $A_2 = \mathsf{san}\_X^{s}_{b\to b'}\ c_2$, for some $c_2 \in b$ such that $c_2 \neq c_1$.

In the first case, the first two items to be shown follow immediately. Since the proof of (item 3) and (item 4c) is equivalent to the one in the second case, we will solely focus on that one.

We can immediately deduce that $A = \mathsf{san}\_X^{s}_{b\to b'}\ x$, where $x\sigma_j = c_j$ for $j \in [1, 2]$.

Furthermore, any rule $\ell_{1i} \in \widetilde{\ell_1}$ must be of the form $\ell_{1i} = \textsc{San}\ (X^{s}_{b\to b'}, c_1, c'_{1i})$ for some $c'_{1i} \in b'$. Furthermore, this implies that $p_{1i} = Pr[X^{s}_{b\to b'}(c_1) = c'_{1i}]$ and $T_{1i} = S_1$ and $B_{1i} = c'_{1i}$.

We choose $\ell_{2i} := \textsc{San}\ (X^{s}_{b\to b'}, c_2, c'_{2i})$, where $c'_{2i} := c'_{1i}$, $p_{2i} := Pr[X^{s}_{b\to b'}(c_2) = c'_{2i}]$ and $T_{2i} := S_2$ and $B_{2i} := c'_{2i}$. It follows immediately that $B_{2i} = c'_{2i} = c'_{1i} = B_{1i}$.

We can easily verify that this fulfills (item 1).

We now show that for every $(\ell_{2i}, p_{2i}, [T_{2i}, B_{2i}])$ for $i \in [1, n]$ the tuple

$$(\ell_{1i}, p_{1i}, [T_{1i}, B_{1i}])$$

satisfies the remaining conditions (item 2), (item 3), and (item 4c).

For the sake of readability, we now fix i for the remainder of this proof and drop it from all the previous indexes.

We know that $T_1 = S_1$ and $T_2 = S_2$. Thus we define $T := S$ as a skeleton store for T_1, T_2. Since $B_1 = B_2$ we choose $B := B_1$ as an appropriate skeleton expression for B_1, B_2.

We know that $c_1, c_2 \in b$ and that $x\sigma_j = c_j$ for $j \in [1, 2]$ and that thus $x :!_l b \in \Gamma'$ for some index l. We select $\Gamma'' := \Gamma'\{x :!_{l-1} b / x :!_l b\}$. Furthermore, we define the substitutions σ_1', σ_2' as follows: $y\sigma_j' := \begin{cases} y\sigma_j & \text{if } y \neq x \\ y\sigma_j & \text{if } y = x \text{ and } l > 1 \\ \bot & \text{otherwise} \end{cases}$

This means, that if $l = 1$ then x is neither contained in the domain of Γ'' nor in the domain of σ_1', σ_2'. If $l > 1$ the substitutions and the domain of Γ' remain unchanged.

In the latter case, (item 2) follows immediately, since the substitutions have not changed and the store is not affected.

In the former case, x is no longer contained in the domain of the constant environment and that of the constant substitutions. Since $B_1 = B_2$ it follows immediately that $B_1 \Bumpeq_{B;\sigma_1';\sigma_2'} B_2$, but proving that $T_1 \Bumpeq_{T;\sigma_1';\sigma_2'} T_2$ requires some more reasoning: we know that $x :!_1 b \in \Gamma'$ and that $\Gamma, \Gamma' \vdash [S, \mathsf{san\_}X^s_{b \to b'}\ x] : \tau$. Since $x \in fv(\mathsf{san\_}X^s_{b \to b'}\ x)$, it must be the case that $x :!_1 b \in \Gamma_A'$, since this binding is needed to type-check the skeleton expression $\mathsf{san\_}X^s_{b \to b'}\ x$. This in particular means that by the definition of environment addition $x \notin dom(\Gamma_C')$ and thus $x \notin fv(S_j)$. This immediately implies that $S\sigma_j = S\sigma_j'$.

We now focus on (item 3).

We know that $\Gamma_A, \Gamma_A' \vdash \mathsf{san\_}X^s_{b \to b'}\ x : \tau$ and that $\Gamma' = \Gamma_C' + \Gamma_A'$. As seen above it must be the case that $x \in dom(\Gamma_A')$. We define $\Gamma_C'' := \Gamma_C'$ and Γ_A'' in such a way that $\Gamma_A' = \Gamma_A'' + x :!_1 b$.

By Lemma B.18 and rule SAN it must be the case that
$\Gamma_A, \Gamma_A' \vdash \mathsf{san\_}X^s_{b \to b'}\ x :!_\infty b'$ and $\Gamma_A, \Gamma_A' \vdash !_\infty b' <: \tau$.

By Lemma B.6 and Lemma B.8 it follows that $\Gamma_A, \Gamma_A'' \vdash !_\infty b' <: \tau$ (note that variable bindings have no influence on subtyping).

We know that $B = c_1' \in b'$. By rule SIG it holds that $\emptyset \vdash B :!_1 b'$. We apply $!_k$I to derive that $\emptyset \vdash B :!_\infty b'$. Using Lemma B.9 this also implies that $\Gamma_A, \Gamma_A'' \vdash B : !_\infty b'$. Applying SUB yields $\Gamma_A, \Gamma_A'' \vdash B : \tau$.

Since $T = S$ and $\Gamma_C, \Gamma_C' = \Gamma_C, \Gamma_C''$ by Definition B.9 it follows that $\Gamma, \Gamma'' \vdash [T, B] : \tau$.

The first part of (item 4c) follows immediately by our selection of $\Gamma'', \sigma_1', \sigma_2'$, Definition B.7 and Definition B.8. The second part follows immediately from the hypothesis that $X^{\epsilon/k}_{b \to b'}$ provides ϵ/k-differential privacy.

We conclude with the only inductive case RED-LET.

Case Red-Let: In this case A_1 must be of the form $\mathsf{let}\ x = P_1\ \mathsf{in}\ Q_1$. Since A_1 and A_2 are (A, σ_1, σ_2)-similar it must be the case that $A_2 = \mathsf{let}\ x = P_2\ \mathsf{in}\ Q_2$ and $A = \mathsf{let}\ x = P\ \mathsf{in}\ Q$ such that $P\sigma_j = P_j$ and $Q\sigma_j = Q_j$ for $j \in [1, 2]$.

By definition of Red-Let this means that $[S_j, A_j] \xrightarrow{\widetilde{\ell_1}}_{\widetilde{p_1}} [T_1, \mathsf{let}\ \widetilde{x = P'_j}\ \mathsf{in}\ Q_j]$, where $[S_j, P_j] \xrightarrow{\widetilde{\ell_1}}_{\widetilde{p_1}} \widetilde{[T_1, P'_j]}$.

We know that $\Gamma_A, \Gamma'_A \vdash A : \tau$. By Lemma B.18 and Let it must be the case that there exist $\Delta_1 \Delta'_1, \Delta_2, \Delta'_2$ such that $\Gamma_A^{(\prime)} = \Delta_1^{(\prime)} + \Delta_2^{(\prime)}$ and

- $\Delta_1, \Delta'_1 \vdash P : \rho_1$,
- $\Delta_2, \Delta'_2, x : \rho_1 \vdash Q : \rho_2$,
- $\Gamma_A, \Gamma'_A \vdash \rho_2 <: \tau$.

for some types ρ_1, ρ_2.

It follows by Definition B.9 that $(\Gamma_{\mathcal{C}}, \Gamma'_{\mathcal{C}}) + (\Delta_1, \Delta'_1) \vdash [S, P] : \rho_1$.

It can be easily seen that $(dom(\Delta'_2) \setminus dom(\Delta'_1)) \cap fv(P) = \emptyset$, since otherwise $\Delta_1, \Delta'_1 \vdash P : \rho_1$ would not have succeeded.

We can thus define the constant substitutions σ_1^*, σ_2^* as

$$x\sigma_j^* := \begin{cases} x\sigma_j^* & \text{if } x \in dom(\Gamma'_{\mathcal{C}}, \Delta'_1) \\ \perp & \text{otherwise} \end{cases}.$$

It is easy to see that $\Gamma'_{\mathcal{C}}, \Delta'_1$ is a constant environment for σ_1^*, σ_2^*. Furthermore, it holds that $[S_1, P_1] \bumpeq_{[S,P];\sigma_1^*;\sigma_2^*} [S_2, P_2]$.

All necessary conditions for applying the induction hypothesis are fulfilled. The statement follows immediately by applying the induction hypothesis and constructing $\Gamma'', \sigma'_1, \sigma'_2$ using similar reasoning as in the beginning of this case.

□

B.1.5 Differential privacy results

The proof of the differential privacy theorem relies on the opponent typability lemma, saying that all opponents are well-typed. As usual in type systems for cryptographic protocols [31], we require that the opponent is annotated with an untrusted type (Un in the literature, $\tau \in \mathcal{OPP}$ in our case), which characterizes the values that can be sent to and received from the opponent. Note that we are not constraining the opponent, since typing annotations do not affect the semantics of expressions.

Definition B.12 (Opponent). *A closed expression is an opponent if it is only annotated with types in $\mathcal{OPP}$.*

In order to type-check the opponent, we further need to make the functions exported by the signature available to the opponent. This can easily be achieved by making all function types exponential, as formalized below.

Definition B.13 (Extended Signature)**.** *Let Σ be a signature. We define Σ^+ as the smallest signature satisfying the following conditions*

1. *if $c : \phi \in \Sigma$, then $c : \phi \in \Sigma^+$*

2. *if $f : \phi \in \Sigma$, then $f : \phi \in \Sigma^+$*

3. *if $f : \phi \in \Sigma$, then $f : [\phi]_\infty \in \Sigma^+$.*

It is easy to show that validity is preserved by extension.

Proposition B.4 (Validity Preservation)**.** *Σ is valid if and only if Σ^+ is valid.*

Throughout this paper, we assume the signature Σ to be valid and we type-check expressions using Σ^+.

Lemma B.22 (Opponent Typability)**.** *For every opponent O, $\emptyset \vdash O :!_\infty(\tau_1 \multimap \tau_2)$ for all $\tau_1, \tau_2 \in \mathcal{OPP}$.*

Proof. By induction on the structure of O, using Lemma B.15 and Lemma B.16. □

By combining Theorem B.1, Lemma B.22, and a counting argument on reduction probabilities, we can finally prove the extended version of Theorem 4.1, that is, all well-typed expressions are ϵ, τ-differentially private. Notice that we actually prove a more general property, parameterized by the noise added in the protocol execution. The second condition in the theorem can be dropped if one considers the system without the exponential mechanism.

Restatement B.1 (of the Generalization of Theorem 4.1)**.** *For all $k \in \mathbb{R}^{>0}$, all types τ,and all closed expressions P such that the following conditions hold:*

- *the parameter of all noise addition primitives occurring in P is set to $s := e^{-\epsilon/k}$ (i.e., they are of the form $\mathsf{add\_noise}^{e^{-\epsilon/k}}_{\mathbb{Z}\to\mathbb{Z}}\ M$)*

- *the parameter of all general mechanism primitives $\mathsf{san\_}X^s_{b_1\to b_2}\ M$ for the mechanism $X^s_{b_1\to b_2}$ occurring in P is set to $s := \epsilon/k$ (i.e., they are of the form $\mathsf{san\_}X^{\epsilon/k}_{b_1\to b_2}\ M$) and the respective mechanism $X^s_{b_1\to b_2}$ provides s-differential privacy*

- *$\emptyset \vdash P : \tau \multimap \rho$ for some $\rho \in \mathcal{OPP}$*

P is $\epsilon/k, \tau$-differentially private.

Proof. In the following proof we let $\mathcal{X}$ denote the finite universe of sanitization mechanisms $X^{s}_{b_1 \to b_2}$ that are used in the general mechanism primitives $\mathsf{san\_}X^{s}_{b_1 \to b_2}\ M$ occurring in P. We assume that $\mathcal{X} = \{(X^1)^s_{W^1}, \ldots, (X^w)^s_{W^w}\}$. Note that if we want to consider protocols that do not make use of the general mechanism primitive $\mathcal{X} = \emptyset$.

By Lemma B.22, $\emptyset \vdash O :!_\infty(\rho \multimap \rho')$ for some $\rho' \in \mathcal{OPP}$. We also know that $\emptyset \vdash P :!_\infty(\tau \multimap \rho)$ and, thus, $\emptyset \vdash O(PD) : \rho'$ and $\emptyset \vdash O(PD') : \rho'$ for all databases D, D' of type τ.

By Theorem B.1, for each execution trace $O(PD) \xrightarrow{\ell_1,\ldots,\ell_m\ \ *}_p 1$, there exists an execution trace $O(PD') \xrightarrow{\ell'_1,\ldots,\ell'_m\ \ *}_{p'} M$, constant substitutions σ, σ', a skeleton N and a constant environment Γ such that $N\sigma = 1$, $N\sigma' = M$, and $\Gamma \vdash N : \rho'$. We know that because $\rho' \in \mathcal{OPP}$ it holds that $\rho' = [\rho']_\infty$. By an inspection of the typing rules for values, it must be the case that $\Gamma = \infty\Gamma'$ for some Γ', i.e., all variables are bound to types with replication index $!_\infty$.

We now show that $\frac{p}{p'} \leq e^{\frac{\epsilon \cdot \delta_\tau(D,D')}{k}}$. Let $p = p_1 \cdot \ldots \cdot p_m$ and $p' = p'_1 \cdot \ldots \cdot p'_m$, where p_i is the probability associated to ℓ_i and p'_i the one associated to ℓ'_i. By Theorem B.1, either

- $\frac{p_i}{p'_i} = 1$ and the distance between matching configurations is preserved by the reduction (item 4a) or
- $\ell_i = \textsc{Noise}\ (z_i, \bar{z}_i, e^{-\epsilon/k})$ and $\ell'_i = \textsc{Noise}\ (z'_i, \bar{z}'_i, e^{-\epsilon/k})$, in which case $\frac{p_i}{p'_i} \leq e^{\frac{\epsilon \cdot \delta_{\mathbb{Z}}(z_i,z'_i)}{k}}$ and the distance between matching configurations is reduced by $\delta_{\mathbb{Z}}(z_i, z'_i)$ (item 4b) or
- $\ell_i = \textsc{San}\ (X^{\epsilon/k}_{\bar{b}\to\bar{\bar{b}}}, \bar{c}_i, \bar{d}_i)$ and $\ell'_i = \textsc{San}\ (X^{\epsilon/k}_{\bar{b}\to\bar{\bar{b}}}, \bar{c}'_i, \bar{d}'_i)$ for some $X^{\epsilon/k}_{\bar{b}\to\bar{\bar{b}}} \in \mathcal{X}$, in which case $\frac{p_i}{p'_i} \leq e^{\frac{\epsilon \cdot \delta_{\bar{b}}(\bar{c}_i,\bar{c}'_i)}{k}}$ and the distance between matching configurations is reduced by $\delta_{\bar{b}}(\bar{c}_i, \bar{c}'_i)$ (item 4c).

Let

- $S := \{i \mid \ell_i = \textsc{Noise}\ (z_i, \bar{z}_i, e^{-\epsilon/k})\}$ and
- $T^j := \{i \mid \ell_i = \textsc{San}\ ((X^j)^{\epsilon/k}_{W^j}, \bar{c}_i, \bar{d}_i)\}$, where $W^j = \bar{b}^j \to \bar{\bar{b}}^j$, for $j \in [1, w]$.

We have that

$$
\begin{aligned}
&\frac{p}{p'}\\
=&\frac{p_1}{p'_1} \cdot \ldots \cdot \frac{p_m}{p'_m}\\
\leq&\prod_{i \in S} e^{\frac{\epsilon \cdot \delta_{\mathbb{Z}}(z_i,z'_i)}{k}} + \prod_{i \in T^1} e^{\frac{\epsilon \cdot \delta_{\bar{b}^1}(\bar{c}_i,\bar{c}'_i)}{k}} + \ldots + \prod_{i \in T^w} e^{\frac{\epsilon \cdot \delta_{\bar{b}^w}(\bar{c}_i,\bar{c}'_i)}{k}}\\
=&e^{\sum_{i \in S} \frac{\epsilon \delta_{\mathbb{Z}}(z_i,z'_i)}{k}} + e^{\sum_{i \in T^1} \frac{\epsilon \delta_{\bar{b}^1}(\bar{c}_i,\bar{c}'_i)}{k}} + \ldots + e^{\sum_{i \in T^w} \frac{\epsilon \delta_{\bar{b}^w}(\bar{c}_i,\bar{c}'_i)}{k}}\\
\leq&e^{\frac{\epsilon \cdot \delta_\tau(D,D')}{k}}
\end{aligned}
$$

where the last inequality is a direct consequence of the inequalities ruling the distance between matching configurations expressed in Theorem B.1 (item 4a, item 4b, and item 4c).

Furthermore, since $\Gamma = \infty\Gamma'$, $\mathsf{dist}_\Gamma(1, M) \overset{\text{def}}{=} \mathsf{dist}_\Gamma(\sigma, \sigma')$ can be finite only if $\sigma = \sigma'$, that is, $M = 1$.

So far, we have reasoned about a single pair of matching execution traces. We have to extend this result to all executions traces leading to 1. Let $\hat{p}_1, \ldots, \hat{p}_n$ be the probabilities of each of the n possible execution traces of $O(PD)$ leading to 1 and let $\hat{p}'_1, \ldots, \hat{p}'_n$ be the probabilities of the matching execution traces of $O(PD')$. For every $i \in [1, n]$, we know that $\frac{\hat{p}_i}{\hat{p}'_i} \leq e^{\frac{\epsilon \cdot \delta_\mathcal{T}(D,D')}{k}}$ by the previous result. We now use Lemma B.19 to conclude that $\frac{\Pr[O(PD) \to^* 1] = \hat{p}_1 + \ldots + \hat{p}_n}{\Pr[O(PD') \to^* 1] = \hat{p}'_1 + \ldots + \hat{p}'_n} \leq e^{\frac{\epsilon \cdot \delta_\mathcal{T}(D,D')}{k}}$, that is, P is ϵ/k,$\mathcal{T}$-differentially private. □

B.2 Soundness of DF7$_{\text{alg}}$

This chapter contains the proof the soundness and completeness of the algorithmic DF7$_{\text{alg}}$ type system that was introduced in Section 4.6.

We first show the that algorithmic subtyping is both sound and complete, which will be needed in the proof of the complete algorithmic system DF7$_{\text{alg}}$.

Lemma B.23 (Soundness and Completeness of Algorithmic Subtyping). *For every $\Gamma, \tau, \tau', \phi, \psi$, the following statements hold:*

1. *If $\Gamma \vdash \tau$, then $\Gamma \vdash_{\mathsf{alg}} \tau <: \tau$.*

2. *If $\Gamma \vdash \tau :: \mathsf{pub}$ and $\Gamma \vdash \tau' :: \mathsf{tnt}$, then $\Gamma \vdash_{\mathsf{alg}} \tau <: \tau'$.*

3. *For all $k, t \in \mathbb{R}^{>0} \cup \{\infty\}$ it holds that if $\Gamma \vdash_{\mathsf{alg}} !_1\phi <: !_1\psi$ and $k \leq t$ then $\Gamma \vdash_{\mathsf{alg}} !_t\phi <: !_k\psi$.*

4. *$\Gamma \vdash \tau <: \tau'$ if and only if $\Gamma \vdash_{\mathsf{alg}} \tau <: \tau'$.*

Proof. 1. Follows immediately by the definition of Sub Refl Alg.

2. We proceed by induction on the structure of τ. If $\tau \neq_{\mathsf{str}} \tau'$ we can immediately conclude by an application of Sub Kind Alg. Otherwise, if τ and τ' share the same top-level constructor we first note that that $\tau = !_\infty\phi$ for some ϕ such that $\Gamma \vdash \phi :: \mathsf{pub}$ by definition of the only applicable kinding rule Kind Pub and $\tau' = !_k\psi$ for some ψ, k such that $\Gamma \vdash \psi :: \mathsf{tnt}$ by definition of the only applicable kinding rule Kind Tnt and ϕ and ψ share the same top-level constructor. We further note that $k \leq \infty$.

Case $\phi = \psi = \alpha$ or $\phi = \psi = b \in \Sigma$: We can immediately conclude using statement (1).

Case $\phi = \rho_1 \otimes \rho_2$ and $\psi = \rho'_1 \otimes \rho'_2$: We know that $\Gamma \vdash \rho_1 :: \mathsf{pub}$ and $\Gamma \vdash \rho_2 :: \mathsf{pub}$ and $\Gamma \vdash_{\mathsf{alg}} \rho'_1 :: \mathsf{tnt}$ and $\Gamma \vdash_{\mathsf{alg}} \rho'_2 :: \mathsf{tnt}$ by KIND PAIR. We can apply the induction hypothesis twice to derive that $\Gamma \vdash \rho_1 <: \rho'_1$ and $\Gamma \vdash \rho_2 <: \rho'_2$. We conclude by an application of SUB PAIR ALG.

Case $\phi = \rho_1 \multimap \rho_2$ and $\psi = \rho'_1 \multimap \rho'_2$: We proceed similar to the previous case, using the definition of KIND FUN, applying the induction hypothesis twice to the results and concluding by an application of SUB FUN ALG.

Case $\phi = \rho_1 + \rho_2$ and $\psi = \rho'_1 + \rho'_2$: We proceed similar to the previous case, using the definition of KIND SUM, applying the induction hypothesis twice to the results and concluding by an application of SUB SUM ALG.

Case $\phi = \mu\alpha.\rho$ and $\psi = \mu\alpha.\rho'$: We can immediately conclude by an application of SUB KIND REC ALG.

3. Proof by case analyis of the last applied subtyping rule in the derivation of $\Gamma \vdash_{\mathsf{alg}} !_1\phi <:_1 \psi$. We first note that $\Gamma \nvdash !_1\phi :: \mathsf{pub}$ by inspection of the kinding rules, since public types must have replication indices ∞. This in particular means that the last applied subtyping rule cannot have been SUB KIND ALG or SUB KIND REC ALG.

   Note that in all remaining cases there exists a premise that compares the replication indices. In the derivation of $\Gamma \vdash_{\mathsf{alg}} !_1\phi <:_1 \psi$ this premise of the last applied rule R ALG will be set to $1 \leq 1$. Since we know that $k \leq t$ we can replace the premise by $k \leq t$ and apply R ALG to derive that $\Gamma \vdash_{\mathsf{alg}} !_t\phi <:_k \psi$.

4. - The direction " $\Gamma \vdash_{\mathsf{alg}} \tau <: \tau' \Rightarrow \Gamma \vdash \tau <: \tau'$" follows by straightforward induction on the derivation of $\Gamma \vdash_{\mathsf{alg}} \tau <: \tau'$. We know that $\tau = !_t\phi$ and $\tau' = !_k\psi$ for some ϕ, ψ, t, k. We inspect the last applied algorithmic typing rule

   Case SUB REFL ALG: We know that $\phi = \psi$ and $k \leq t$. Using SUB REFL we can show that $\Gamma \vdash !_1\phi <: !_1\psi_1$ and we can conclude that $!_t\phi <: !_k\psi$ by an application of SUB REPL.

   Case SUB PAIR ALG: We know that $\phi = \rho_1 \otimes \rho_2$ and $\psi = \rho'_1 \otimes \rho'_2$ and $\Gamma \vdash_{\mathsf{alg}} \rho_1 <: \rho_2$ and $\Gamma \vdash_{\mathsf{alg}} \rho'_1 <: \rho'_2$ and $k \leq t$. We can apply the induction hypothesis twice to derive that $\Gamma \vdash \rho_1 <: \rho_2$ and $\Gamma \vdash \rho'_1 <: \rho'_2$. We conclude by first applying SUB PAIR and then SUB REPL.

   Case SUB FUN ALG: Similar to the previous case, by applying the induction hypothesis twice and concluding by an application of SUB FUN and SUB REPL.

   Case SUB SUM ALG: Similar to the previous cases, by applying the induction hypothesis twice and concluding by an application of SUB SUM and SUB REPL.

   Case SUB POS REC ALG: Similar to the previous cases, by applying the induction hypothesis to the subtyping premise of the rule and concluding by an application of SUB POS REC and SUB REPL.

Case Sub Kind Rec Alg: We can immediately conclude using Sub Kind.

Case Sub Kind Alg: We can immediately conclude using Sub Kind.

- The direction " $\Gamma \vdash \tau <: \tau' \Rightarrow \Gamma \vdash_{\mathsf{alg}} \tau <: \tau'$" is proven by induction on the derivation of $\Gamma \vdash \tau <: \tau'$. We distinguish the following cases, depending on the last applied rule:

Case Sub Refl: We conclude by statement (1).

Case Sub Kind: We conclude by statement (2).

Case Sub Repl: We apply the induction hypothesis to the premise and conclude by statement (3).

Case Sub Pair: We know that $\tau =!_1(\rho_1 \otimes \rho_2)$ and $\tau' =!_1(\rho_1' \otimes \rho_2')$ for some $\rho_1, \rho_2, \rho_1', \rho_2'$ and $\Gamma \vdash \rho_1 <: \rho_1'$ and $\Gamma \vdash \rho_2 <: \rho_2'$. We can apply the induction hypothesis twice to derive that $\Gamma \vdash_{\mathsf{alg}} \rho_1 <: \rho_1'$ and $\Gamma \vdash_{\mathsf{alg}} \rho_2 <: \rho_2'$. Since $1 \leq 1$ we can conclude by an application of Sub Pair Alg.

Case Sub Fun: The case is similar to the previous one, we apply the induction hypothesis to both premises of Sub Fun and conclude by an application of Sub Fun Alg, using the fact that $1 \leq 1$.

Case Sub Sum: The case is similar to the previous ones, we apply the induction hypothesis to both premises of Sub Sum and conclude by an application of Sub Sum Alg, using the fact that $1 \leq 1$.

Case Sub Pos Rec: We know that $\tau =!_1(\mu\alpha.\rho)$ and $\tau' =!_1(\mu\alpha.\rho')$ and $\Gamma, \alpha \vdash \rho <: \rho'$ for some ρ, ρ' and α occurs only positively in ρ, ρ'. In particular, by the only possibly applicable kinding rule Kind Pub we know that $\Gamma \nvdash !_1(\mu\alpha.\rho) :: \mathsf{pub}$, since public types must have replication indices ∞. We can apply the induction hypothesis to derive that $\Gamma, \alpha \vdash_{\mathsf{alg}} \rho <: \rho'$ and conclude by an application of Sub Pos Rec Alg, using the fact that $1 \leq 1$.

□

Lemma B.24 (Algorithmic Weakening). *For every $\Gamma, A, \tau, \Gamma', \Delta$ such that $\Gamma \vdash A : \tau; \Gamma'$ and $\Gamma + \Delta \vdash \diamond$, we have that $\Gamma + \Delta \vdash A : \tau; \Gamma' + \Delta$.*

Lemma B.25 (Derived Algorithmic Judgments). *If $\Gamma \vdash_{\mathsf{alg}} A : \tau; \Gamma'$ then $\Gamma' \vdash \tau$ and $dom(\Gamma) = dom(\Gamma')$.*

We now have all the ingredients to prove the soundness and completeness of DF7$_{\text{alg}}$. In the following, we write $\langle A \rangle$ to denote the typing environment obtained by removing the typing annotations from A.

Restatement B.2 (of Theorem 4.2). *For every Γ, A, and τ, the following conditions hold:*

1. *If $\Gamma \vdash A : \tau$ then there exist Γ', A' such that $\Gamma \vdash_{\mathsf{alg}} A' : \tau; \Gamma'$ and $A = \langle A' \rangle$.*
2. *If $\Gamma \vdash_{\mathsf{alg}} A : \tau; \Gamma'$ then there exists Γ'' such that $\Gamma'' \vdash \langle A \rangle : \tau$ and $\Gamma = \Gamma' + \Gamma''$.*

Proof. We first prove item 1. The proof proceeds by induction on the length of the derivation of $\Gamma \vdash A : \tau$ (i.e., on the number of expression typing rules therein). We first discuss the base cases:

Case Var $[A := x,\ \tau :=!_1\phi,\ \Gamma := \Delta, x :!_k\phi]$ We know that $\Delta, x :!_k\phi \vdash x :!_1\phi$ is derived by $\Gamma, x :!_k\phi \vdash \diamond$ and $k \geq 1$. We set $A' := x$ and $\Gamma' = \Delta, x :!_{k-1}\phi$. Since $A = A' = x$, $\langle A' \rangle = A$. By Var, we obtain $\Gamma \vdash_{\mathsf{alg}} A' : \tau; \Gamma'$.

Case Sig $[A := M,\ \tau :=!_1\phi]$ We set $A' := A$ and $\Gamma' := \Gamma$. The proof is straightforward, since the hypotheses of Sig and Sig Alg coincide.

Case Read The proof is similar to the one for Sig.

Case Read Opp The proof is similar to the one for Sig.

Case Write The proof is similar to the one for Sig.

Case Ref The proof is similar to the one for Sig.

Case Ref Opp The proof is similar to the one for Sig.

We now discuss the induction step:

Case $\otimes$I $[A := (M_1, M_2),\ \tau :=!_1(\tau_1 \otimes \tau_2),\ \Gamma := \Delta_1 + \Delta_2]$ We know that $\Delta_1 + \Delta_2 \vdash (M_1, M_2) :!_1(\tau_1 \otimes \tau_2)$ is proved by $\Delta_1 \vdash M_1 : \tau_1$ and $\Delta_2 \vdash M_2 : \tau_2$. By induction hypothesis, there exist Δ'_1, M'_1 such that $\Delta_1 \vdash_{\mathsf{alg}} M'_1 : \tau; \Delta'_1$ and $\langle M'_1 \rangle = M_1$, and there exist Δ'_2, M'_2 such that $\Delta_2 \vdash_{\mathsf{alg}} M'_2 : \tau; \Delta'_2$ and $\langle M'_2 \rangle = M_2$.

By Lemma B.3 and Lemma B.2, $\Delta_1 + \Delta_2 \vdash \diamond$. By Lemma B.24, $\Delta_1 + \Delta_2 \vdash_{\mathsf{alg}} M'_1 : \tau; \Delta'_1 + \Delta_2$. By Lemma B.25 and Lemma B.3, $\Delta_2 + \Delta'_1 \vdash \diamond$ (we recall that $\Delta_2 + \Delta'_1 = \Delta'_1 + \Delta_2$). By Lemma B.24, $\Delta_2 + \Delta'_1 \vdash_{\mathsf{alg}} M'_2 : \tau; \Delta'_2 + \Delta'_1$.

We set $A' := (M'_1, M'_2)$ and $\Gamma' := \Delta'_2 + \Delta'_1$. By $\otimes$I Alg, we obtain $\Gamma \vdash A' : \tau; \Gamma'$.

Case $+/\mu$I Straightforward, by induction hypothesis.

Case $\multimap$I $[A := \lambda x.B,\ \tau :=!_1(\rho \multimap \rho')]$ We know that $\Gamma \vdash A : \tau$ is proved by $\Gamma, x : \rho \vdash B : \rho'$. By induction hypothesis, there exist Δ, B' such that $\Gamma, x : \rho \vdash B' : \rho'; \Delta$ and $\langle B' \rangle = B$.

We set $A' := \lambda x : \rho.B'$ and $\Gamma' := \Delta \backslash x$. By $\multimap$I Alg, $\Gamma \vdash A' : \tau; \Gamma'$.

Sub Straightforward, by the induction hypothesis and Lemma B.23.

Case !I Straightforward, by induction hypothesis.

Case $\otimes$E $[\Gamma = r\Delta + \Delta',\ A := \mathsf{let}\ (x, y) = M\ \mathsf{in}\ B]$ We know that $r\Delta + \Delta' \vdash \mathsf{let}\ (x, y) = M\ \mathsf{in}\ B : \tau$ is proved by $\Delta \vdash M :!_1(\tau_1 \otimes \tau_2)$ and $\Delta', x :!_r\tau_1, y :!_r\tau_2 \vdash B : \tau$.

By induction hypothesis, there exists Δ'' such that $\Delta \vdash_{\mathsf{alg}} M' :!_1(\tau_1 \otimes \tau_2); \Delta''$ and $\langle M' \rangle = M$. By !I Alg, $r\Delta \vdash_{\mathsf{alg}} M'_{!r} :!_r(\tau_1 \otimes \tau_2); r\Delta''$.

By Lemma B.3 and Lemma B.2, $r\Delta + \Delta' \vdash \diamond$. By Lemma B.24, $r\Delta + \Delta' \vdash_{\mathsf{alg}} M'_{!r} :!_r(\tau_1 \otimes \tau_2); r\Delta'' + \Delta'$.

By induction hypothesis, there exists Δ''' such that $\Delta', x :!_r\tau_1, y :!_r\tau_2 \vdash \langle B\rangle : \tau; \Delta'''$. By Lemma B.25 and Lemma B.3, $\Delta' + r\Delta'' \vdash \diamond$. By Lemma B.3, we also know that $\Delta', x :!_r\tau_1, y :!_r\tau_2 \vdash \diamond$ and, thus, $\Delta', x :!_r\tau_1, y :!_r\tau_2 + r\Delta'', x :!_0\tau_1, y :!_0\tau_2 \vdash \diamond$. By Lemma B.24, $(\Delta', x :!_r\tau_1, y :!_r\tau_2) + (r\Delta'', x :!_0\tau_1, y :!_0\tau_2) \vdash \langle B\rangle : \tau; \Delta''' + (r\Delta'', x :!_0\tau_1, y :!_0\tau_2)$. Notice that $(\Delta', x :!_r\tau_1, y :!_r\tau_2) + (r\Delta'', x :!_0\tau_1, y :!_0\tau_2) = (\Delta' + r\Delta''), x :!_r\tau_1, y :!_r\tau_2$.

We set $\Gamma' := (\Delta'''\backslash x)\backslash y + r\Delta''$.

By $\otimes$E ALG, $\Gamma \vdash_{\mathsf{alg}} \mathsf{let}\ (x, y) = M'_{!r}\ \mathsf{in}\ \langle B\rangle : \tau; \Gamma'$.

Case +E The proof is similar to the one for $\otimes$E.

Case μE The proof is similar to the one for $\otimes$E.

Case $\multimap$I The proof is similar to the one for $\otimes$E.

Case LET The proof is similar to the one for $\otimes$E.

Case ADD-NOISE Straightforward, by induction hypothesis.

Case SAN Straightforward, by induction hypothesis.

Case EQ The proof is similar to the one for $\otimes$E.

We now prove item 2. The proof proceeds by induction on the length of the derivation of $\Gamma \vdash_{\mathsf{alg}} A : \tau; \Gamma$ (i.e., on the number of expression typing rules therein). We first discuss the base cases:

Case VAR ALG [$A := x$, $\tau :=!_1\phi$, $\Gamma := \Delta, x :!_k\phi$, $\Gamma' := \Delta, x :!_{k-1}\phi$] We know that $\Delta, x :!_k\phi \vdash_{\mathsf{alg}} x :!_1\phi; \Delta, x :!_{k-1}\phi$ is proved by $\Delta, x :!_k\phi \vdash \diamond$ and $k \geq 1$.

We set $\Gamma'' := \Delta, x :!_1\phi$. Notice that $\Gamma = \Gamma' + \Gamma''$. We can easily see that $\Delta, x :!_1\phi \vdash \diamond$. By VAR, $\Gamma'' \vdash \langle A\rangle : \tau$.

Case SIG ALG [$A := M$, $\tau :=!_1\phi$, $\Gamma' := \Gamma$] Set $\Gamma'' := \emptyset$. The result follows from SIG, observing that $\emptyset \vdash \diamond$.

Case READ ALG The proof is similar to the one for SIG ALG.

Case READ OPP ALG The proof is similar to the one for SIG ALG.

Case WRITE ALG The proof is similar to the one for SIG ALG.

Case REF ALG The proof is similar to the one for SIG ALG.

Case REF OPP ALG The proof is similar to the one for SIG ALG.

We now discuss the induction step:

Case $\otimes$I Alg $[A := (M_1, M_2),\ \tau :=!_1(\tau_1 \otimes \tau_2)]$ We know that

$$\Gamma \vdash_{\mathsf{alg}} (M_1, M_2) :!_1(\tau_1 \otimes \tau_2); \Gamma'$$

is proved by $\Gamma \vdash_{\mathsf{alg}} M_1 : \tau_1; \Delta$ and $\Delta \vdash_{\mathsf{alg}} M_2 : \tau_2; \Gamma'$.

By induction hypothesis, there exists Δ' such that $\Delta' \vdash \langle M_1 \rangle : \tau_1$ and $\Gamma = \Delta + \Delta'$. By induction hypothesis, there exists Δ'' such that $\Delta'' \vdash \langle M_2 \rangle : \tau_2$ and $\Delta = \Gamma' + \Delta''$. We choose $\Gamma'' := \Delta' + \Delta''$. By $\otimes$I, $\Gamma'' \vdash \langle (M_1, M_2) \rangle :!_1(\tau_1 \otimes \tau_2)$. We also have that $\Gamma = \Delta + \Delta' = \Gamma' + \Delta' + \Delta'' = \Gamma' + \Gamma''$.

Case $+/\mu$I Alg Straightforward, by induction hypothesis.

Case $\multimap$I Alg $[A := \lambda x : \rho.B,\ \tau :=!_1(\rho \multimap \rho'),\ \Gamma' : \Delta\backslash x]$ We know that $\Gamma \vdash_{\mathsf{alg}} \lambda x : \rho.B :!_1(\rho \multimap \rho'); \Delta\backslash x$ is proved by $\Gamma, x : \rho \vdash_{\mathsf{alg}} B : \rho'; \Delta$. By induction hypothesis, there exists Γ'', ρ'' such that $\Gamma'', x : \rho'' \vdash \langle B \rangle : \rho'$ and $\Gamma, x : \rho = \Delta + \Gamma'', x : \rho''$. We set $\Gamma'' := \Gamma''$. By $\multimap$ I, $\Gamma'' \vdash \lambda x.B :!_1(\rho \multimap \rho')$. Notice that $\Gamma = \Gamma' + \Gamma''$.

Case !I Alg $[A := M,\ \tau :=!_k\phi,\ \Gamma := k\Delta,\ \Gamma' := k\Delta']$ We know that $k\Delta \vdash_{\mathsf{alg}} M_{!k} : !_k\phi; k\Delta'$ is proved by $\Delta \vdash_{\mathsf{alg}} M :!_1\phi; \Delta'$. By induction hypothesis, there exists Δ'' such that $\Delta'' \vdash \langle M \rangle :!_1\phi$ and $\Delta = \Delta' + \Delta''$. We set $\Gamma'' = k\Delta''$. By !I, we get $k\Delta'' \vdash \langle M \rangle :!_k\phi$. Notice that $\Gamma = k\Delta = k\Delta' + k\Delta'' = \Gamma' + \Gamma''$.

Case Sub Alg Straightforward, by the induction hypothesis and Lemma B.23.

Case $\otimes$E Alg $[A := \mathsf{let}\ (x, y) = M_{!r}\ \mathsf{in}\ B,\ \Gamma' := (\Delta\backslash x)\backslash y]$ We know that $\Gamma \vdash_{\mathsf{alg}} \mathsf{let}\ (x, y) = M_{!r}\ \mathsf{in}\ B : \tau; (\Delta\backslash x)\backslash y$ is proved by $\Gamma \vdash_{\mathsf{alg}} M_{!r} :!_r(\tau_1 \otimes \tau_2); \Delta'$ and $\Delta', x :!_r\tau_1, y :!_r\tau_2 \vdash_{\mathsf{alg}} B : \tau; \Delta$.

By induction hypothesis, there exists Δ'' such that $\Delta'' \vdash \langle M \rangle :!_r(\tau_1 \otimes \tau_2)$ and $\Gamma = \Delta' + \Delta''$. By an inspection of the typing rules, this means that $\Delta'' = r\Delta'''$ and $\Delta''' \vdash \langle M \rangle :!_1(\tau_1 \otimes \tau_2)$.

By induction hypothesis, there exists Δ'''' such that $\Delta'''' \vdash \langle B \rangle : \tau$ and $\Delta', x : !_r\tau_1, y :!_r\tau_2 = \Delta + \Delta''''$. We set $\Gamma'' := r\Delta''' + ((\Delta''''\backslash z)\backslash y)$.

By $\otimes$E, we get $\Gamma'' \vdash \mathsf{let}\ (x, y) = \langle M \rangle\ \mathsf{in}\ \langle B \rangle : \tau$.

Notice that $\Gamma = \Delta' + \Delta'' = (((\Delta\backslash x)\backslash y) + ((\Delta''''\backslash x)\backslash y)) + r\Delta''' = \Gamma' + \Gamma''$.

Case +E Alg $[A := \mathsf{case}\ M_{!r}\ \mathsf{of}\ x\ \mathsf{in}\ B_1\ \mathsf{else}\ B_2\ ,\ \Gamma' := \mathsf{min}(\Delta_1, \Delta_2)\backslash x]$ We know that $\Gamma \vdash_{\mathsf{alg}} \mathsf{case}\ M_{!r}\ \mathsf{of}\ x\ \mathsf{in}\ B_1\ \mathsf{else}\ B_2\ : \tau; \mathsf{min}(\Delta_1, \Delta_2)\backslash x$ is proved by $\Gamma \vdash_{\mathsf{alg}} M_{!r} :!_r(\tau_1 + \tau_2); \Delta$, $\Delta, x :!_r\tau_1 \vdash_{\mathsf{alg}} B_1 : \tau; \Delta_1$, and $\Delta, x :!_r\tau_2 \vdash_{\mathsf{alg}} B_2 : \tau; \Delta_2$.

By induction hypothesis, there exists Δ' such that $\Delta' \vdash \langle M \rangle :!_r(\tau_1 \otimes \tau_2)$ and $\Gamma = \Delta + \Delta'$. By an inspection of the typing rules, this means that $\Delta' = r\Delta''$ and $\Delta'' \vdash \langle M \rangle :!_1(\tau_1 \otimes \tau_2)$.

By induction hypothesis, there exist Δ'_1, Δ'_2 such that $\Delta'_1 \vdash \langle B_1 \rangle : \tau$, $\Delta'_2 \vdash \langle B_2 \rangle : \tau$, $\Delta, x :!_r\tau_1 = \Delta_1 + \Delta'_1$, and $\Delta, x :!_r\tau_2 = \Delta_2 + \Delta'_2$. We set $\Gamma'' := r\Delta'' + \mathsf{max}(\Delta'_1, \Delta'_2)\backslash x$, where max is defined analogously to min.

By $\otimes$E and Lemma B.9, we get $\Gamma'' \vdash \mathsf{case}\ \langle M \rangle\ \mathsf{of}\ x\ \mathsf{in}\ \langle B_1 \rangle\ \mathsf{else}\ \langle B_2 \rangle\ : \tau$. Notice that $\Gamma = \Delta + \Delta' = (\Delta_1 + \Delta_1')\backslash x + r\Delta'' = (\Delta_2 + \Delta_2')\backslash x + r\Delta''$=$\mathsf{min}(\Delta_1, \Delta_2)\backslash x + \mathsf{max}(\Delta_1', \Delta_2')\backslash x + r\Delta''$=$\Gamma' + \Gamma''$.

Case μE ALG The proof is similar to the one for +E ALG.

Case $\multimap$E ALG The proof is similar to the one for $\otimes$E ALG.

Case LET ALG The proof is similar to the one for $\otimes$E ALG.

Case ADD-NOISE ALG Straightforward, by induction hypothesis.

Case SAN ALG Straightforward, by induction hypothesis.

Case EQ ALG The proof is similar to the one for $\otimes$E ALG.

□