Thesis for The Degree of Doctor of Philosophy

On the Foundations of Information-Flow Control and Effects

Carlos Tomé Cortiñas

CHALMERS

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2024

On the Foundations of Information-Flow Control and Effects
Carlos Tomé Cortiñas

© Carlos Tomé Cortiñas, 2024

ISBN 978-91-8103-022-8
Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 5480
ISSN 0346-718X

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice
Gothenburg, Sweden 2024

A mi abuela Marisa (in memoriam)

y

a mis abuelos Maruja y Manuel

Abstract

There is no doubt that society depends crucially on software systems. Correct software is therefore a pressing matter. An important aspect of software correctness is security: for example, a banking application is secure, if at least it does not send your credit card number to an unauthorised third party. It is well-known that software is riddled with bugs. Some are introduced by human mistakes; some by deficiencies of the programming languages at use. Security vulnerabilities arising from both kinds of bugs appear time and again, affecting the security needs of millions of users. It thus becomes imperative to design programming languages that help programmers avoid bugs. Developers must demand assurances that secure programming languages indeed produce secure software. This warrants mathematical justification in the form of proof. To increase reusability, these proofs would ideally be carried out in a framework that is modular on the features of the programming language.

Language-based security investigates the security aspects of software, such as confidentiality or integrity, from the perspective of programming language theory. One approach to information-flow control (IFC) studies the design of programming languages in which programs are secure-by-construction. In this approach, types guide the development of programs, and the type system ensures that well-typed programs are secure. Practical programs feature effects such as nontermination or printing. Type systems for IFC need therefore to take these into account to guarantee security. Establishing the correctness of IFC type-systems for effectful languages is a complex matter, which often relies on ad hoc methods that depend on the concrete kinds of effects.

In this thesis, we describe a novel theory of information flow with effects. In our framework the correctness of IFC type-systems can be proved modularly with respect to the kind of effects. This theory builds upon existing models of information flow and models of effects, and thus, is readily applicable. Independently, this thesis makes two additional contributions. First, we show how to extend concurrent IFC languages with asynchronous exceptions, which, e.g., enable secure interthread communication. Second, we present a new technique for proving correctness of IFC type systems based on normalization. As a byproduct of independent interest, we present novel normalization results for the family of so-called Fitch-style modal calculi.

Keywords security, programming languages, information-flow control

Acknowledgements

I would like to start by thanking my supervisor Alejandro Russo. Supervising me has probably not been the easiest thing, and doing so through a pandemic hasn't helped either. I want to thank you for our multiple collaborations, and for helping me bring this PhD thesis to an end.

A special thanks goes to my, first and foremost, friends, but also colleagues, collaborators, and coauthors, Nachiappan Valliappan and Fabian Ruch. This PhD thesis would definitely have not been the same without you. I will cherish our endless discussions and their usual conclusion asking "what does it mean?". These have impacted the kind of researcher I aspire to be.

A huge thanks goes to my colleagues and friends at the department for providing such a friendly and fun work environment. Thanks Irene, Víctor, Abhiroop, Matthí, Sandro, Ivan, Mohammad, Benjamin, Robert, Agustín, Ale (el otro), Elisabet, Prabhat, Lorenzo, and Victor. If you feel that I have forgotten you, then your name probably deserves to be in this list, my apologies.

Life in in Gothenburg these last five years would not have been as enjoyable, if it wasn't for all the people that were there to support me during this journey; sometimes even despise long distances. Thank you Μαρία, Ειρήνη, Antonio, Duarte, Arturo, Rachele, Klara and the many of you whose name didn't make it in here due to space constraints!

My final thanks goes to my family. I cannot begin to express my gratitude towards them for their unconditional love and support throughout these years. Thanks to my mother Élida for always looking at things from the bright side, and to my father Eduardo for sparking my curiosity for science when I was little. Gracias mamá y papá.

Contents

Overview

Introduction

Information security is at stake. Computing systems that manage users' sensitive data are vulnerable to attacks, sometimes innocent,[1] by individuals or organizations who want to steal their data. Unfortunately, the success of attacks is higher that anyone would hope for, thus compromising the information security needs of these systems and their users.

The primary reason for the success of these attacks is that the languages and tools upon which computing systems are built have not been designed with security in mind. Therefore, they are unable to guarantee that users' sensitive information is handled in a secure fashion.

This thesis investigates the theoretical foundations of programming languages in which secure-by-construction systems ought to be written. The overarching goal of this thesis is to provide better tools and methods to help in the design and implementation of programming languages in which programs handling sensitive data are *secure-by-construction*. To this end, this thesis makes several novel contributions to the research area known as language-based security (LBS), specifically to information-flow control (IFC).

Language-Based Security Language-based security (LBS) (Kozen 1999; Schneider et al. 2001; Sabelfeld and Myers 2003) is a research area that approaches security in computing systems from the perspective of programming language theory. The goal of language-based security is to guarantee that several aspects of information security, such as confidentiality or integrity, are preserved at the level of computer programs. To that end, language-based security studies programming languages, type systems, runtime monitors, and other mechanisms that constrain how programs can *access* and *use* sensitive data. These mechanisms enforce that all accesses and uses happen in concordance with a security

[1] Bugs are present in virtually every piece of software.

property that acts as a formal baseline for secure programs. Security properties further depend on higher-level specifications, known as security policies, that specify what and how principals, those users who partake in the system, can access and use sensitive information.

Before spelling out the concrete contributions of this thesis, we need a way to organise them. For this purpose, let us recall Schenider's definition[2] of language-based security as:

> A set of techniques based on programming language theory and implementation, including semantics, types, optimization, and verification, brought to bear on the security question.

This rather broad definition alludes to, among others, *types* and *semantics* as studied in the theory of programming languages. From the point of view of this thesis, these two subareas consist of:

Types Refers to the use of type disciplines—viz. type systems—for enforcing at compile time that programs preserve information security. Those type systems typically come equipped with special-purpose type constructors to label pieces of programs and data with *security levels*. Then it is the job of the typing rules to guarantee that when the program runs labelled data behaves according to the security policy. Examples of these kinds of (security) type systems are the dependency core calculus (Abadi et al. 1999) and the Haskell library MAC (Russo 2015).

Semantics Refers to the use of mathematical models, so called denotational semantics, to give mathematical meaning to programs, security specifications, and security properties. In addition, those models can be used to study and verify security properties of security type systems and languages. Examples of models for information security are partial equivalence relations (Sabelfeld and Sands 2001), dependency categories (Abadi et al. 1999), and classified sets (Kavvos 2019).

This categorization serves as a conceptual framework in which to organise the articles contained in this thesis according to their main area of contribution, namely types and semantics.

Thesis Structure This thesis consists of five peer-reviewed articles—A, B, C, D and E—some of which have been accepted for publication in conferences, symposia, and workshops spanning from security- to programming languages

[2]Attributed by Kozen (1999, Section 4).

-oriented venues. As mentioned above, we classify these articles according to their area of contribution. Figure I.1 depicts this classification.

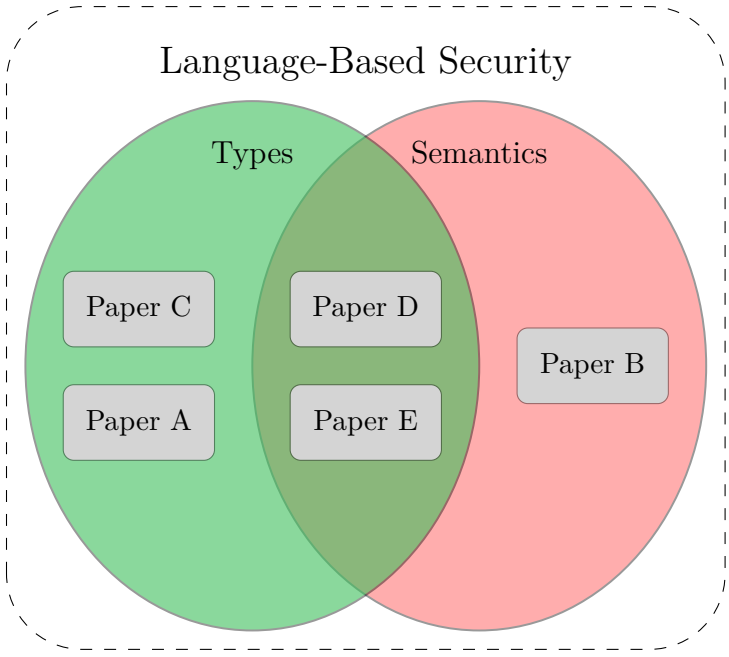


Figure I.1.: Areas of contribution of the articles included in this thesis.

Rest of this Introduction The rest of this chapter is divided into two sections:

- Section I.1 introduces information-flow control. This language-based security technique is the focus of most of the articles in this thesis. The aim of this section is not to introduce new material, but to present relevant terminology and concepts in a unified form.
- Section I.2 outlines the contents of each article contained in this thesis, and discusses its main contributions. This section further states the individual contributions of the author of this thesis.

I.1. Information-Flow Control

Information-flow control is a language-based security technique that aims to secure information by tracking *where* users' sensitive data propagates during

program execution. In contrast to other language-based security techniques such as access control, information-flow control guarantees *end-to-end* security. The classic scenario for information-flow control consists of a system with public and secret information, such that information should not propagate from secret to public. Formally, we call this specification a security policy. Then the objective of information-flow control is to ensure that a program's public output is independent of its secret inputs, that is, it satisfies a security—or information-flow—property known as noninterference (Goguen and Meseguer 1982). This condition defines what programs are secure. Information-flow control mechanisms usually enforce noninterference by tracking how information propagates through the control flow of programs, ensuring that their public outputs do not depend on their secret inputs. Enforcement can be done statically at compile time, dynamically at runtime, or combining both.

To summarize, information-flow control approaches to security consist of:

- a programming language;
- a framework for describing security policies;
- a framework for writing security specifications for programs;
- a formal security property expressing what does it mean for programs to be secure with respect to security specifications; and
- an enforcement mechanism to ensure that programs comply with their security specifications, i.e. they are secure.

In the rest of this section, we describe some of the points above in more detail.

Security Policies

Security policies express in high-level languages the information security requirements of computing systems, that is, how information is allowed to be used in those systems (Kozyri et al. 2022).

In this thesis, we consider security policies that consist of a set of principals, i.e. those users or actors whose sensitive data is at stake, together with a specification of the permitted and forbidden flows of information among these. The intuitive reading is that these policies specify what flows of information are allowed through the execution of programs handling principals' data. Note that security policies are independent of the programming language under consideration, and hence of the programs one can write. In Sections I.1 and I.1, we will discuss the missing pieces that relate security policies to programs.

Example I.1.1. Consider a system with three principals, say Alice, Bob and Charlie. These could, for instance, be employees of some company. A security policy would then include at least these principals: `Alice`, `Bob` and `Charlie`; together with a specification of allowed information flows that depends on the information security concerns of the company.

For example, suppose that Charlie is the boss of Alice and Bob. If Alice and Bob were working on separate projects and they were not allowed to share information, then the security policy would specify that information is permitted to flow from `Alice` and `Bob` to `Charlie` but neither from `Alice` to `Bob` nor from `Bob` to `Alice`. It is implicitly understood that information is always allowed to flow from a principal to themselves.

Typically, security policies are formalized using some kind of lattice (Denning 1976). The elements of this lattice, or security levels, sometimes also called labels, or sensitivities, are an abstraction over the distinct principals (or sets of them), and the partial order relation specifies the allowed flows of information. In fact, the lack of a relationship between two levels corresponds to a forbidden flow. Note that security levels in a lattice and "real" principals need not be in one-to-one correspondence; for instance, in the previous example there could be a security level `Alice and Bob` for data that is permitted to flow both to Alice and to Bob, although there is no principal "Alice and Bob".

An intuitive way to understand a security policy is by visualizing its Hasse diagram. The diagram representing the security policy in Example I.1.1 is the following:

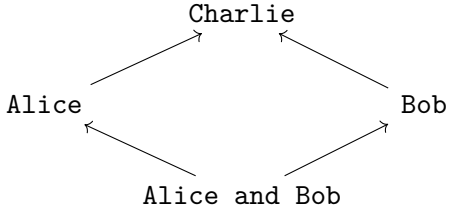


Figure I.2.: Hasse diagram representing the security policy in Example I.1.1.

Security policies can be as complex as the situation requires. Sometimes these may even consist of an infinite number of security levels, e.g., to model open systems. In the simplest scenario, the security policy consists of two security levels `L`, for *public*, and `H`, for *secret*, and a relation states that every flow of information is allowed except from `H` to `L`, i.e. flows to equal or higher

sensitivity are allowed, e.g. from public to secret, but flows from higher to lower sensitivity are forbidden, namely from secret to public. The Hasse diagram representing this policy is the following:

Figure I.3.: Hasse diagram representing the two-level security policy.

The public-secret situation generalizes to more complex scenarios, i.e., those with nontrivial security lattices. In those cases, the attacker belongs to an arbitrary security level "public", and confidential data to a level "secret". Importantly, flows of information from the later to the former should be disallowed.

Security Specifications

Security specifications classify different parts of programs, such as input and return values, routines and functions, input and output channels, effects, etc., with security levels drawn from a security policy.

In this thesis, we focus on static approaches to information-flow control via type systems, where the security levels of different parts of programs are known at compile time. These are ascribed security levels using special-purpose type constructors.

Example I.1.2. A security specification for a program of type $f : \mathsf{Bool} \to \mathsf{Bool}$ (in some typed lambda calculus) might be written as $f : \mathsf{Bool}^{\mathtt{H}} \to \mathsf{Bool}^{\mathtt{L}}$. This specification states that the security level of its Boolean input is H (i.e. secret) and that of its Boolean output is L (i.e. public).

Information-Flow Properties

Information-flow properties establish a formal relation between security policies, security specifications, and programs that handle sensitive data. An information-flow property is, given a security policy and a security specification, a predicate on programs. This predicate defines what it means for programs to be *secure*.

In this thesis, we are concerned with information-flow properties derived from noninterference, a concept introduced by Goguen and Meseguer in their

seminal work on security policies and models (Goguen and Meseguer 1982). In their words, noninterference states that:

> One group of users, using a certain set of commands is *noninterfering* with another group of users if what the first group does with those commands has no effect on what the second group of users can see.

In the context of information-flow control and this thesis, we rephrase their definition in a more modern dressing as:

> Secret data should not influence what can be observed from the public outputs of a program.

Formally,

Definition I.1.1 (Noninterference)**.** A program satisfies *noninterference* if its public outputs are independent of its secret inputs.

What constitutes a program's secret inputs and public outputs, and hence what is the precise statement of noninterference, varies with respect to

- the features of the programming language, and thus what behaviour programs can exhibit, e.g. general recursion, printing, memory references;
- the security policy and the security specification; and,
- the observational power of the attacker and the permitted amount of information leakage, i.e. what they can infer from observing the execution of programs, e.g. leakage from differences among termination and divergence, timing output events, inspecting the intermediate contents of memory.

Note that Definition I.1.1 presupposes that the so-called attacker—against who we wish to secure the system—has security level public. The attacker is therefore privy to observe the public outputs that programs produce.

To formally prove that a program satisfies noninterference, one usually considers the public outputs that the program produces in several executions for distinct secret input values. If varying the secret input does not affect the public outputs of the program, that is, the public outputs agree on all executions, then it must be the case that those outputs are truly independent of the secret inputs, and hence the program is noninterferent.

Enforcement Mechanisms

Enforcement mechanisms ensure that programs comply with their security specifications. These mechanisms can roughly be divided into two categories, *static* and *dynamic*, according to whether the enforcement happens before the execution of programs, i.e. at compile time, or during their execution, i.e. at runtime.

- Dynamic enforcement mechanisms usually consist of runtime monitors that are executed along the program itself. These monitors halt or modify the execution of the program if they detect that some flow of information is about to violate the security policy.

- Static enforcement mechanisms analyse programs at compile time to determine whether all the flows of information are permitted by a given security specification (and thus the security policy). Static analyses for information-flow control appear in several forms. On the traditional side, a static analyser takes a program as input and outputs whether the program is secure or not. More modern approaches build the analysis into the programming language's type system. In these approaches, security is equated to well-typedness in the sense that all well-typed programs are secure.

  In this thesis, we exclusively focus on (static) type systems for information-flow control.

Within these two extremes, there is a broad spectrum of mechanisms that combine aspects of both static and dynamic enforcement. These are called *hybrid* mechanisms.

I.2. Thesis Contributions

This thesis is a compendium of articles. In this section we briefly outline the contents of each article and state the individual contributions of the author. Reformatted versions of these articles appear in subsequent chapters of this thesis.

I.2.1. Pure Information-Flow Control with Effects Made Simple

In this paper we introduce a novel primitive distr to modularly extend type systems for (effect-free) IFC—e.g. dependency core calculus (DCC) (Abadi et al. 1999)—with effects in a principled manner. To evidence the modularity

of our approach, we present several extensions of the sealing calculus (SC) by Shikuma and Igarashi (2008) with effects. These extensions consist of naively combining SC with a graded monad for effects in the style of Katsumata (2014). In the resulting system, effects and IFC a priori do not interact: many secure programs whose effects depend on sensitive data are not expressible. Our key insight is that adding a primitive, which we call distr, is enough to regain the expressiveness of state-of-the-art IFC languages with effects.

In this work we showcase this idea by considering three extensions of SC with different effects, namely printing, (first-order) global store, and a combination of both. We prove that the languages resulting from these extensions are secure, i.e. they satisfy suitable versions of noninterference. We further present a Haskell library as a proof-of-concept implementation. The languages, i.e. their syntax and operational semantics, and their respective security guarantees have been mechanized in the proof assistant Agda.

Statement of Contributions This paper was coauthored with Alejandro Russo. Carlos devised the idea of modularizing languages for effectful information-flow control (IFC) via the primitive distr. Carlos formalized and mechanized the calculi presented in the paper. Carlos wrote most of the paper.

This paper is currently unpublished. A version of the manuscript appears as Chapter A of this thesis.

I.2.2. Information Flow and Effects Via Distributive Laws

In this paper, we present a semantic justification that the primitive distr (introduced in Chapter A) is in some sense the best solution to extend type systems for IFC with effects in a modular way. For that purpose, we investigate semantics of effects, such as side effects, nondeterminism, and printing, and their interaction with the usual operation of "labelling" in categories of classified sets (Kavvos 2019). These well-known models of information flow have been used to prove noninterference properties for effect-free IFC calculi such as DCC (see Kavvos (2019)).

In this work we show that computational monads on classified sets are suitable to model "indistinguishability" of observable effects, and hence, to encode information flow properties of effectful programs. That is, for an effect (exceptions, for instance) given as a computational monad on the category of sets, we show that distinct monads on classified sets encode distinct security properties of effectful programs, e.g. exception-sensitive and exceptions-insensitive noninterference. We provide extensive evidence of this claim by considering information-flow monads for the usual effects: side effects, store, nondeterminism,

exceptions, or divergence.

Information-flow monads for effects and redaction—the semantic counterpart of labelling—a priori do not interact: it is not valid to eliminate redacted values to computations. In this work, we show that the validity of this elimination principle corresponds to a distributive law from redaction to the monad for effects. We prove, in fact, that once effects and their intended security properties are specified by a monad on classified sets, then it is a property of redaction and this monad whether a (necessarily unique) distributive law exists. That is, either redaction and this monad interact nicely or they do not, and there is nothing we can do to fix it. Of course, one can choose a different monad, but this would correspond to considering a different information-flow property for effectful programs.

Statement of Contributions This paper is the sole work of Carlos.

This paper is currently unpublished. The latest version of the manuscript appears as Chapter B of this thesis.

I.2.3. Securing Asynchronous Exceptions

In this paper we present MACASYNC, an extension of the concurrent IFC language MAC (Vassena et al. 2018) with asynchronous exceptions. Asynchronous exceptions—sometimes called interrupts—are a useful language feature that enables threads to communicate and interact with one another and with the runtime system. Asynchronous exceptions can be used, for instance, to program timeouts, speculative execution patterns, and user interrupts. (Marlow et al. 2001) In principle, naively combining asynchronous exceptions with existing IFC features (e.g. synchronization variables) opens up new possibilities for information leakage.

In this work, we to show how to incorporate asynchronous exceptions to MAC in a secure fashion. To that end, we extend MAC with new primitives to i) allow threads to refer to other threads, ii) send asynchronous exceptions to other threads, and iii) selectively disable receiving asynchronous exceptions in certain parts of programs, e.g. in critical zones. We further adapt the operational semantics of MAC to account for asynchronous exceptions, following the work by Marlow et al. (2001). Lastly, we prove that MACASYNC inherits MAC's strong security properties, namely progress-sensitive noninterference (Vassena et al. 2018). The language, i.e. its syntax and operational semantics, and the security guarantees have been mechanized in the proof assistant AGDA (Abel et al. 2005–).

Statement of Contributions This paper was coauthored with Marco Vassena and Alejandro Russo. Carlos suggested the idea of studying asynchronous exceptions in the context of IFC. Carlos devised the calculus and its formal semantics, and mechanized them along with the security guarantees in Agda. Carlos wrote the technical sections of the paper.

This paper has been published in the 33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020. A reformatted version appears as Chapter C of this thesis.

I.2.4. Simple Noninterference by Normalization

In this paper we present a novel proof technique for proving noninterference. This technique exploits the insight that to reason about information-flow properties of a programming language it is enough to consider a restricted (and simple) class of programs. If programs in this class satisfy the property of interest, and this class represents all programs in a suitable sense, then all programs satisfy the property.

Based on this idea, in this work we show a new proof of noninterference for a static information-flow control calculus based on a terminating fragment of the calculus that formalizes the Haskell information-flow control library SecLib (Russo et al. 2008). The calculus is a simply typed lambda calculus extended with unit, product, and sum types, and a family of monads for each security level. The simple class of programs, from which a noninterference property follows rather directly, is characterized as the normal forms of a standard equational theory which includes among others β-, η-, and δ-equations for monadic types.

Specifically, the paper describes in detail i) the class of normal forms, ii) a procedure, based on normalization by evaluation (NbE), for obtaining normal forms for arbitrary terms, iii) a proof that normal forms indeed faithfully represent classes of equivalent terms, and iv) a proof of noninterference based on analysing the normal forms. The results in this paper have been mechanized in the proof assistant Agda (Abel et al. 2005–).

Statement of Contributions This paper was coauthored with Nachiappan Valliappan. Both Nachiappan and Carlos contributed equally to the formalization, the Agda mechanization, and the writing of the paper.

This paper has been published in the Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, CCS 2019, London, United Kingdom, November 11-15, 2019. A reformatted version appears as Chapter D of this thesis.

I.2.5. Normalization for Fitch-Style Modal Calculi

In this paper, we present a modular approach to semantic proofs of normalization for Fitch-style modal lambda calculi (Borghuis 1994), which enables programming with necessity modalities in typed lambda calculi. These calculi extend the usual notion of typing context with a delimiting operator, denoted by a lock, and can easily incorporate different modal axioms. These axioms correspond to programming operations on modal types.

In this work, we leverage the possible-world semantics of Fitch-style calculi to yield a modular proof of normalization via NbE. We show that NbE models can be constructed for calculi that incorporate the axioms K, T and 4 of modal logic, as suitable instantiations of the possible-world semantics. Finally, we showcase several consequences of normalization for proving meta-theoretic properties of Fitch-style calculi (Clouston 2018) as well as programming language-oriented applications such as capability safety (Choudhury and Krishnaswami 2020) and information flow based on different interpretations of the necessity modality. The key results in this paper have been mechanized in the proof assistant Agda (Abel et al. 2005–).

Statement of Contributions This paper was coauthored with Nachiappan Valliappan and Fabian Ruch. Carlos contributed to the technical content and writing of the section on applications of normalization to proving meta-theoretic properties of programming language-oriented applications. Carlos also helped writing Section 3 of the paper.

This paper has been published in the Proc. ACM Program. Lang. Vol. 6, ICFP, 2022. A reformatted version appears as Chapter E of this thesis.

Bibliography

Abadi, Martín et al. (1999). "A Core Calculus of Dependency". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, pp. 147–160. DOI: 10.1145/292540.292555. URL: https://doi.org/10.1145/292540.292555 (cit. on pp. 4, 10).

[SW] Abel, Andreas et al., *Agda 2* 2005–. Chalmers University of Technology and Gothenburg University. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php (cit. on pp. 12–14).

Borghuis, Valentijn Anton Johan (1994). *Coming to terms with modal logic*. On the interpretation of modalities in typed λ-calculus, Dissertation, Technische Universiteit Eindhoven, Eindhoven, 1994. Technische Universiteit Eindhoven, Eindhoven, pp. x+219 (cit. on p. 14).

Choudhury, Vikraman and Neel Krishnaswami (2020). "Recovering purity with comonads and capabilities". In: *Proc. ACM Program. Lang.* 4.ICFP, 111:1–111:28. DOI: 10.1145/3408993. URL: https://doi.org/10.1145/3408993 (cit. on p. 14).

Clouston, Ranald (2018). "Fitch-Style Modal Lambda Calculi". In: *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Christel Baier and Ugo Dal Lago. Vol. 10803. Lecture Notes in Computer Science. Springer, pp. 258–275. DOI: 10.1007/978-3-319-89366-2_14. URL: https://doi.org/10.1007/978-3-319-89366-2%5C_14 (cit. on p. 14).

Denning, Dorothy E. (1976). "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5, pp. 236–243. DOI: 10.1145/360051.360056. URL: https://doi.org/10.1145/360051.360056 (cit. on p. 7).

Goguen, Joseph A. and José Meseguer (1982). "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, pp. 11–20. DOI: 10.1109/SP.1982.10014. URL: https://doi.org/10.1109/SP.1982.10014 (cit. on pp. 6, 8, 9).

Katsumata, Shin-ya (2014). "Parametric effect monads and semantics of effect systems". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, pp. 633–646. DOI: 10.1145/2535838.2535846. URL: https://doi.org/10.1145/2535838.2535846 (cit. on p. 11).

Kavvos, G. A. (2019). "Modalities, cohesion, and information flow". In: *Proc. ACM Program. Lang.* 3.POPL, 20:1–20:29. DOI: 10.1145/3290333. URL: https://doi.org/10.1145/3290333 (cit. on pp. 4, 11).

Kozen, Dexter (1999). "Language-Based Security". In: *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings*. Ed. by Miroslaw Kutylowski et al. Vol. 1672. Lecture Notes in Computer Science. Springer, pp. 284–298. DOI: 10.1007/3-540-48340-3_26. URL: https://doi.org/10.1007/3-540-48340-3%5C_26 (cit. on pp. 3, 4).

Kozyri, Elisavet et al. (2022). "Expressing Information Flow Properties". In: *Found. Trends Priv. Secur.* 3.1, pp. 1–102. DOI: 10.1561/3300000008. URL: https://doi.org/10.1561/3300000008 (cit. on p. 6).

Marlow, Simon et al. (2001). "Asynchronous Exceptions in Haskell". In: *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*. Ed. by Michael Burke and Mary Lou Soffa. ACM, pp. 274–285. DOI: 10.1145/378795.378858. URL: https://doi.org/10.1145/378795.378858 (cit. on p. 12).

Russo, Alejandro (2015). "Functional pearl: two can keep a secret, if one of them uses Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, pp. 280–288. DOI: 10.1145/2784731.2784756. URL: https://doi.org/10.1145/2784731.2784756 (cit. on p. 4).

Russo, Alejandro et al. (2008). "A library for light-weight information-flow security in haskell". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, pp. 13–24. DOI: 10.1145/1411286.1411289. URL: https://doi.org/10.1145/1411286.1411289 (cit. on p. 13).

Sabelfeld, Andrei and Andrew C. Myers (2003). "Language-based information-flow security". In: *IEEE J. Sel. Areas Commun.* 21.1, pp. 5–19. DOI: 10.1109/JSAC.2002.806121. URL: https://doi.org/10.1109/JSAC.2002.806121 (cit. on p. 3).

Sabelfeld, Andrei and David Sands (2001). "A Per Model of Secure Information Flow in Sequential Programs". In: *High. Order Symb. Comput.* 14.1, pp. 59–91. DOI: 10.1023/A:1011553200337. URL: https://doi.org/10.1023/A:1011553200337 (cit. on p. 4).

Schneider, Fred B. et al. (2001). "A Language-Based Approach to Security". In: *Informatics - 10 Years Back. 10 Years Ahead.* Ed. by Reinhard Wilhelm. Vol. 2000. Lecture Notes in Computer Science. Springer, pp. 86–101. DOI: 10.1007/3-540-44577-3_6. URL: https://doi.org/10.1007/3-540-44577-3%5C_6 (cit. on p. 3).

Shikuma, Naokata and Atsushi Igarashi (2008). "Proving Noninterference by a Fully Complete Translation to the Simply Typed Lambda-Calculus". In: *Log. Methods Comput. Sci.* 4.3. DOI: 10.2168/LMCS-4(3:10)2008. URL: https://doi.org/10.2168/LMCS-4(3:10)2008 (cit. on p. 11).

Tomé Cortiñas, Carlos and Nachiappan Valliappan (2019). "Simple Noninterference by Normalization". In: *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, CCS 2019,*

London, United Kingdom, November 11-15, 2019. Ed. by Piotr Mardziel and Niki Vazou. ACM, pp. 61–72. DOI: 10.1145/3338504.3357342. URL: https://doi.org/10.1145/3338504.3357342 (cit. on p. 13).

Tomé Cortiñas, Carlos, Marco Vassena, et al. (2020). "Securing Asynchronous Exceptions". In: *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. IEEE, pp. 214–229. DOI: 10.1109/CSF49147.2020.00023. URL: https://doi.org/10.1109/CSF49147.2020.00023 (cit. on p. 13).

Valliappan, Nachiappan et al. (2022). "Normalization for fitch-style modal calculi". In: *Proc. ACM Program. Lang.* 6.ICFP, pp. 772–798. DOI: 10.1145/3547649. URL: https://doi.org/10.1145/3547649 (cit. on p. 14).

Vassena, Marco et al. (2018). "MAC A verified static information-flow control library". In: *Journal of Logical and Algebraic Methods in Programming* 95, pp. 148–180. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2017.12.003. URL: https://www.sciencedirect.com/science/article/pii/S235222081730069X (cit. on p. 12).

Papers

A

Pure Information-Flow Control with Effects Made Simple

Carlos Tomé Cortiñas and Alejandro Russo

Manuscript

Abstract Information-flow control (IFC) is a promising technology to protect data confidentiality. The foundational work on the dependency core calculus (DCC) positions monads as a suitable abstraction for enforcing IFC. Pure functional languages with effects, like Haskell, can provide IFC as a library (e.g. MAC and LIO), a minor task compared to implementing compilers for IFC from scratch.
Previous works on IFC as a library introduce ad hoc primitives to type programs whose effects do not depend on the sensitive data in context. In this work, we start afresh and ask ourselves: what would we need to extend an effect-free language for IFC with secure effects? The answer turns out to be elegant and simple. In a pure language with effects there is a natural place where information flows from sensitive data to effects need to be restricted, and when effects are tracked in a fine-grained fashion, for instance, with a graded monad, then a *single* primitive is enough to allow secure flows! To support our insight, we present and prove secure several IFC enforcement mechanisms based on extensions of the sealing calculus (SC) with effects using a graded monad. Effects that depend on sensitive data are secured through a novel primitive distr. Our security guarantees are mechanized in the Agda proof assistant. Moreover, we provide an implementation of these mechanisms as a new Haskell library for IFC. Our implementation amounts to less than 10 lines of code for the effect-free part and less than 30 lines for the part with effects. Lastly, we demonstrate that our library is capable of encoding previous Haskell libraries for static IFC.

A.1. Introduction

Information-flow control (IFC) (Sabelfeld and Myers 2003; Hedin and Sabelfeld 2012) is a promising technology to protect data confidentiality. Many IFC approaches are designed to prevent sensitive data from influencing what attackers can observe from a program's public behaviour—a security property known as noninterference (Goguen and Meseguer 1982). IFC mechanisms specify the sensitivity of data via *labels*, and then enforce security by controlling that the flows of information abide by the security policy. In the simplest scenario, there are two labels (alternatively, security levels or sensitivities) `L`, for *public*, and `H`, for *secret*, and the security policy specifies that every flow is allowed except from `H` to `L`—i.e. flows from more to less sensitivity are forbidden. In static approaches to IFC, the sensitivity of data is known a priori, e.g. by specifying it using types, and the enforcement statically decides, e.g. during type checking, whether the program will leak information upon execution. To maintain confidentiality of data, IFC mechanisms need to protect against two kinds of potentially malicious flows: 1) *explicit flows*, when public data directly depends on secret data; and 2) *implicit flows*, when the control flow and public outputs of the program are indirectly influenced by secret data, e.g. due to branching on a `H`-labelled Boolean.

In recent years, the use of pure functional languages has been proliferating for tackling different IFC challenges (e.g. Vassena, Russo, Garg, et al. (2019), Parker et al. (2019), Polikarpova et al. (2020), and Rajani and Garg (2020)). From a pragmatic perspective, pure functional languages can provide IFC security via libraries (Li and S. Zdancewic 2006; Russo et al. 2008; Stefan et al. 2011), which is less demanding than building compilers from scratch (e.g. Simonet (2003) and Myers et al. (2006)).

Pure languages are particularly well suited for controlling information flows because of their abstraction facilities and strong encapsulation of effects. For instance, the popular dependency core calculus (DCC) (Abadi et al. 1999) utilizes the abstract type $\mathsf{T}_{\mathtt{H}}\ A$ to label pieces of data of type A with sensitivity `H` and then the type system ensures that data can only be eliminated into—or flow to—data of equal or higher sensitivity. DCC's security guarantees ensure that programs without effects are secure. A different strand of work aims to provide security in pure languages with effects by restricting the interplay between sensitive data and public effects (e.g. LIO (Stefan et al. 2011), MAC (Russo 2015) and HLIO (Buiras et al. 2015)). In a pure language like Haskell, the *only* programs that can produce effects and thereby interact with the external world have to be of type $\mathsf{IO}\ A$, for some type A. In this light, in order to protect against implicit flows through effects it is enough to control which programs of

IO type are safe to execute.

Let us consider the Haskell library MAC as an example. MAC replaces the IO monad with a custom monad MAC_l of computations that is indexed by a type-level label l. The label l has two purposes: (i) akin to DCC, it is an *upper bound* on the sensitivity of the information "going into" the monad, as well as (ii) it is a *lower bound* on the observers' effects—restricting information "leaving" the monad. Concretely, a computation of type $\mathsf{MAC}_{\mathtt{L}}\,\mathsf{Bool}$ *cannot* branch on secret values but can perform public and secret effects; in contrast, a computation of type $\mathsf{MAC}_{\mathtt{H}}\,\mathsf{Bool}$ can branch on secret data but *cannot* perform public effects.

However, not everything in the garden is rosy. The label l in the MAC monad MAC_l does too many things at once. This leads to situations where the programmer needs to go through some contortions or use "special purpose" primitives like join[1]: $\mathsf{MAC}_l\,\mathsf{Unit} \Rightarrow \mathsf{MAC}_{l'}\,\mathsf{Unit}$ (restricted to $l' \sqsubseteq l$) (Vassena, Buiras, et al. 2016). To illustrate this point, we extend the two-point security policy with two incomparable labels $\mathtt{A}$, for *Alice*, and $\mathtt{B}$, for *Bob*, such that $\mathtt{L} \sqsubseteq \mathtt{A} \sqsubseteq \mathtt{H}$ and $\mathtt{L} \sqsubseteq \mathtt{B} \sqsubseteq \mathtt{H}$ *but* neither $\mathtt{A} \sqsubseteq \mathtt{B}$ nor $\mathtt{B} \sqsubseteq \mathtt{A}$ (the relation $\sqsubseteq$ specifies the permitted flows). With that in mind, let us consider the following program in MAC which receives an $\mathtt{A}$-sensitive Bool, i.e. $\mathit{alice\_sec} : \mathsf{MAC}_{\mathtt{A}}\,\mathsf{Bool}$,[2] and prints a string on the $\mathtt{A}$-observable channel $\mathtt{Ch}_{\mathtt{A}}$:

$$\begin{aligned}&\mathit{prog}_1 : \mathsf{MAC}_{\mathtt{A}}\,\mathsf{Bool} \Rightarrow \mathsf{MAC}_{\mathtt{A}}\,\mathsf{Unit}\\ &\mathit{prog}_1\ \mathit{alice\_sec} = \mathit{alice\_sec} \gg\!= \lambda b.\ \mathsf{if}\ b\ \mathsf{then}\ \mathsf{print}_{\mathtt{Ch}_{\mathtt{A}}}\ \texttt{"Alice is here!"}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \mathsf{else}\ \mathsf{return}\ \mathsf{unit}\end{aligned}$$

In the above program, the information flows according to the security policy (from the $\mathtt{A}$-sensitive value to Alice's channel)—i.e. the program is secure. Consider now a different program that combines prog_1 with printing on the channel $\mathtt{Ch}_{\mathtt{B}}$:

$$\begin{aligned}&\mathit{prog}_2 : \mathsf{MAC}_{\mathtt{A}}\mathsf{Bool} \Rightarrow \mathsf{MAC}\ \boxed{?}\ \mathsf{Unit}\\ &\mathit{prog}_2\ \mathit{alice\_sec} = \mathit{prog}_1\ \mathit{alice\_sec} \gg\!= \lambda\, x.\ \mathsf{print}_{\mathtt{Ch}_{\mathtt{B}}}\ \texttt{"Hi Bob"}\end{aligned}$$

Clearly, prog_2 is still secure since the decision to print to Bob's channel and what is printed does not depend on the contents of the $\mathtt{A}$-sensitive value $\mathit{alice\_sec}$. What label then should replace $\boxed{?}$ in its type? First, the types of the computations on both sides of the bind ($\gg\!=$) have to match. By (i) and (ii) the type of $\mathit{prog}_1\ \mathit{sec}$ has to be $\mathsf{MAC}_{\mathtt{A}}\,\mathsf{Unit}$, and by (ii) $\mathsf{print}_{\mathtt{Ch}_{\mathtt{B}}}\ \texttt{"Hi Bob"}$ has to

[1] The type of join is more general but this simplified form suffices for our purposes.

[2] To the reader familiar with MAC: our point applies equally if one uses the $\mathsf{Labeled}_l$ type to protect sensitive Booleans.

be of type $\mathsf{MAC}_{\mathtt{B}}\,\mathsf{Unit}$. There is a type mismatch! Since the label `L` is a lower bound of `A` and `B`, we can use MAC's join to fix the program:

$$\begin{aligned}&\mathit{prog}_2 : \mathsf{MAC}_{\mathtt{A}}\,\mathsf{Bool} \Rightarrow \mathsf{MAC}_{\mathtt{L}}\,\mathsf{Unit}\\&\mathit{prog}_2\ \mathit{alice\_sec} = \mathsf{join}(\mathit{prog}_1\ \mathit{alice\_sec}) \gg\!= \lambda\, x.\ \mathsf{join}(\mathsf{print}_{\mathtt{Ch}_{\mathtt{B}}}\ \texttt{"Hi Bob"})\end{aligned}$$

Now, something unexpected happened. To assign a type to prog_2, which only mentions labels `A` and `B`, we *have to* resort to another label `L`. What is worse, in the type of prog_2 only the label `A` appears and does so in an argument position which means we know nothing about the program's possible effects. The design decision of indexing the monad MAC by a *single* label, while enabling a simple implementation of IFC as a library (cf. (Russo 2015)), requires the application of special purpose primitives (like join) to mitigate over-approximations of how information flows in and out of computations.

In this work, we take a step back and ask ourselves: what would it take to allow arbitrary effects in a pure language with an already existing mechanism for effect-free IFC? The answer turns out to be elegant and simple. We observe that enforcing IFC in a pure language with effects can be reduced to the *single* point—which we will explain below—where effects and sensitive data interact. Based on this observation, we present a novel IFC mechanism which is arguably simpler than existing IFC libraries, allows to assign more natural types to programs; and overcomes the programming contortions discussed above.

To briefly present our idea, let us assume that effect-free IFC is achieved using a DCC-style abstract type $\mathsf{T}_l\,A$, and we have at our disposal a more refined type constructor Eff of effectful programs akin to IO but annotated with concrete information about *observable* effects. For example, Eff could be annotated with the set of channels where the program might print, the set of exceptions the program might raise, or the set of memory references the program might modify. Moreover, we assume that the security policy specifies the sensitivity of each effect.

Consider, for instance, a scenario with two output channels, namely $\mathtt{Ch}_{\mathtt{L}}$ and $\mathtt{Ch}_{\mathtt{H}}$, that are assigned sensitivities `L` and `H`, respectively. In this scenario, a program $\mathit{prog}_3 : \mathsf{T}_{\mathtt{A}}\,(\mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{B}}\}}\,\mathsf{Unit})$ is a computation that might print to `B`'s channel $\mathtt{Ch}_{\mathtt{B}}$ depending on `A`-sensitive data. This is where sensitive data and effects interact! Assuming $\mathit{alice\_sec} : \mathsf{T}_{\mathtt{A}}\,\mathsf{Bool}$ is in scope, prog_3, for instance,

could have the following implementation:[3]

$$
\begin{aligned}
&\mathit{prog}_3 : \mathsf{T}_{\mathtt{A}}\,(\mathsf{Eff}_{\{\mathtt{Ch_B}\}}\,\mathsf{Unit})\\
&\mathit{prog}_3 = \mathsf{bind}\,b = \mathit{alice\_sec}\\
&\qquad \mathsf{in}\ \mathsf{return}_{\mathtt{A}}(\mathsf{if}\ b\ \mathsf{then}\ \mathsf{print}_{\mathtt{Ch_B}}\ \texttt{"So it was true, huh"}\\
&\qquad\qquad \mathsf{else}\ \mathsf{return}\ \mathsf{unit})
\end{aligned}
$$

In order to run the effects of the inner computation of type $\mathsf{Eff}_{\{\mathtt{B}\}}\,\mathsf{Unit}$ we have to "extract" it first from the $\mathsf{T}_{\mathtt{A}}$ value—recall that the language is pure. In general, extracting anything from $\mathsf{T}_{\mathtt{A}}$ is prohibited as it would render the IFC enforcement unsound: in prog_3, the computation of type $\mathsf{Eff}_{\{\mathtt{Ch_B}\}}\,\mathsf{Unit}$ would become executable, and if the $\mathtt{A}$-sensitive Boolean $\mathit{alice\_sec}$ is true, it will print $\texttt{"So it was true, huh"}$ on Bob's channel $\mathtt{Ch_B}$—a flow that violates the security policy. On the contrary, let us consider a program of a different type, $\mathit{prog}_4 : \mathsf{T}_{\mathtt{A}}\,(\mathsf{Eff}_{\{\mathtt{Ch_H}\}}\,\mathsf{Unit})$. A possible implementation of prog_4 could be:

$$
\begin{aligned}
&\mathit{prog}_4 : \mathsf{T}_{\mathtt{A}}\,(\mathsf{Eff}_{\{\mathtt{Ch_H}\}}\,\mathsf{Unit})\\
&\mathit{prog}_4 = \mathsf{bind}\,b = \mathit{alice\_sec}\\
&\qquad \mathsf{in}\ \mathsf{return}_{\mathtt{A}}(\mathsf{if}\ b\ \mathsf{then}\ \mathsf{print}_{\mathtt{Ch_H}}\ \texttt{"It is true!"}\\
&\qquad\qquad \mathsf{else}\ \mathsf{print}_{\mathtt{Ch_H}}\ \texttt{"It is false!"})
\end{aligned}
$$

In this program, the computation, i.e. $\mathsf{if}\ b\ \mathsf{then}\ \mathsf{print}_{\mathtt{Ch_H}}\ \texttt{"It is true!"}\ \mathsf{else}\ \mathsf{print}_{\mathtt{Ch_H}}\ \texttt{"It is false!"}$, is safe to run since it prints $\mathtt{A}$-sensitive data on the channel $\mathtt{Ch_H}$—a flow permitted by the security policy. In fact, the underlying computation of any program of type $\mathsf{T}_{\mathtt{A}}\,(\mathsf{Eff}_{\{\mathtt{Ch_H}\}}\,\mathsf{Unit})$ is *definitely* safe to run since the only possible effects it might produce are printing to the channel $\mathtt{Ch_H}$, which we know from the type $\mathsf{Eff}_{\{\mathtt{Ch_H}\}}\,\mathsf{Unit}$, and the decision on what to print depends on data of at most sensitivity $\mathtt{A}$, which we know from the type constructor $\mathsf{T}_{\mathtt{A}}$. Securing effectful programs then amounts to allowing any such program prog_4 to extract the computation of type $\mathsf{Eff}_{\{\mathtt{Ch_H}\}}\,\mathsf{Unit}$ from $\mathsf{T}_{\mathtt{A}}$ and run its effects while forbidding prog_3 from doing so. To achieve this, we introduce a novel primitive distr which systematically permits computations to be extracted, and hence, executed only when they are known to depend on data less sensitive than the sensitivity of their observers. With distr we can turn prog_4 into an executable program: $\mathsf{distr}(\mathit{prog}_4) : \mathsf{Eff}_{\{\mathtt{Ch_H}\}}\,\mathsf{Unit}$. We have reduced the enforcement of IFC in pure languages with effects to a *single* primitive; this gives us *modularity*, *clarity* and *simplicity* in the language design and its possible implementations.

[3] bind is DCC's eliminator for the type T_l.

To conclude, we provide an alternative implementation of $prog_2$ with a more natural type using the primitive distr:

$$
\begin{aligned}
&prog_2' : \mathsf{T}_{\mathtt{A}}\ \mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{\mathtt{Ch_A},\mathtt{Ch_B}\}}\ \mathsf{Unit}\\
&prog_2'\ alice\_sec = \mathsf{distr}\ (\mathsf{bind}\ b = alice\_sec\\
&\qquad\qquad\qquad\qquad \mathsf{in\ return}(\mathsf{if}\ b\ \mathsf{then}\ \mathsf{print}_{\mathtt{Ch_A}}\ \texttt{"Alice is here!"}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else\ return\ unit}))\\
&\qquad\qquad\qquad \gg\!= \lambda\, x.\ \mathsf{print}_{\mathtt{Ch_B}}\ \texttt{"Hi Bob"}
\end{aligned}
$$

Our Contributions In this work, we show that IFC in the context of pure languages with effects can be achieved through the combination of the following features: 1. an enforcement for effect-free IFC, 2. a type for tracking observable effects in a fine-grained fashion, and 3. a primitive distr which selectively permits to execute effectful computations which depend on sensitive data.

We present our idea through an IFC enforcement mechanism in the form of a security-type system for programs written in the programming language λ_2, which is a call-by-name variant of the simply-typed λ-calculus (STLC) extended with recursive functions and Booleans. The security-type system for λ_2, which we dub SC, is an adaptation of the sealing calculus (SC) of Shikuma and Igarashi (2008), which is more expressive than DCC (cf. (Tse and S. Zdancewic 2004; Shikuma and Igarashi 2008)). We then extend λ_2 in two different directions by adding printing and global store effects in the form of explicit graded monadic effects (Katsumata 2014). We name these extensions λ_2^{PRINT} and λ_2^{STORE}, respectively. Although the orthogonal extensions of SC for printing and global store readily enforce IFC by not permitting any interaction between labelled values and effects, we additionally extend SC with our novel primitive distr. This allows to type check more of those effectful programs that are secure.

Along with informal argumentations as to why our idea yields secure IFC enforcement mechanisms for the different effects, we prove that programs satisfy suitable versions of noninterference. To increase confidence, we have mechanized these proofs in the Agda proof assistant (Abel, Allais, et al. 2005–). Our proofs are based on the technique of logical relations.

Finally, we realize our idea in the form of a proof-of-concept Haskell library which we call SCLib. The conciseness of our implementation illustrates the elegance and simplicity of our insight: less than 10 lines of code for the effect-free fragment and less than 30 for the part with effects. In order to implement the effect-free IFC mechanism we also present a novel implementation of SC using an encoding of contextual information as type-level capabilities. Our library is at least as expressive as previous work on libraries for IFC in Haskell, which we evidence by showing implementations of SecLib (Russo et al. 2008), DCC

(in its presentation by Algehed (2018)) and MAC in terms of SCLib's interface.

In summary, the technical contributions of this paper are:

- A reformulation of SC a as security-type system, SC, for λ_2 (Section A.2)
- Two extensions of λ_2 and SC for enforcing IFC in pure languages with effects via graded monads. As examples, we consider printing (Section A.3.1) and global store (Section A.3.2) effects
- We present distr, a single primitive that can control the interaction of sensitive data and effects
- Security guarantees and proofs of noninterference based on logical relations for all the enforcements (Section A.4)
- A Haskell implementation using a novel encoding of contextual information as capabilities together with evidence that SCLib can encode existing monadic security libraries (Section A.5)
- Mechanized proofs of our security guarantees as Agda code.

A.2. Effect-Free Information-Flow Control

In this section, we briefly recall the sealing calculus (SC) (Shikuma and Igarashi 2008), introduce the programming language on which we wish to enforce IFC, and explain our adaptation of SC as a security-type system.

The Sealing Calculus SC utilizes an abstract type $\mathsf{S}_l\ A$ for protecting sensitive data.[4] A value of type $\mathsf{S}_l\ A$ is "sealed" at sensitivity l in the sense that it is only available to observers with sensitivity at least as high as l. Values of type $\mathsf{S}_l\ A$ are introduced and eliminated using the primitives seal_l and unseal_l. SC enforces IFC by restricting in which *contexts* a sealed value can be "unsealed". For example, the $\mathtt{A}$-sensitive Boolean $sec :: \mathsf{S}_{\mathtt{A}}\ \mathsf{Bool}$ can only be unsealed in contexts of at least sensitivity $\mathtt{A}$.

Let us illustrate SC with a program that receives two Booleans with sensitivities $\mathtt{A}$ and $\mathtt{B}$, respectively, and computes their conjunction with sensitivity $\mathtt{H}$ ($\wedge : \mathsf{Bool} \times \mathsf{Bool} \Rightarrow \mathsf{Bool}$ implements conjunction):

$$
\begin{aligned}
&and' : \mathsf{S}_{\mathtt{A}}\ \mathsf{Bool} \Rightarrow \mathsf{S}_{\mathtt{B}}\ \mathsf{Bool} \Rightarrow \mathsf{S}_{\mathtt{H}}\ \mathsf{Bool} \\
&and' = \lambda\, sb_1\ sb_2.\, \mathsf{seal}_{\mathtt{H}}\ (\wedge(\mathsf{unseal}_{\mathtt{A}}\ sb_1), (\mathsf{unseal}_{\mathtt{B}}\ sb_2))
\end{aligned}
$$

[4] In the original presentation, the authors use the notation $[A]^l$ instead.

In the above program, the term $\mathsf{seal}_{\mathtt{H}}$ provides the context in which sb_1 and sb_2 can be unsealed: the term $\mathsf{unseal}_{\mathtt{A}}\ sb_1$, for instance, is only well-typed because its label $\mathtt{A}$ flows to $\mathtt{H}$, which is the highest label.

The Language λ_2 is a call-by-name variant of the STLC with Unit and Bool as the only primitive types. We work directly with intrinsically-typed terms, and thus we consider that typing derivations $\Gamma \vdash t : a$ *are* terms. The small-step semantics is specified by a relation $t \to t'$ on closed terms, i.e. $\cdot \vdash t : a$, and we call values those terms t for which $t \not\to$. For reference we include the complete definition in Appendix A.I. Since λ_2 is well-understood we omit any further explanations.

The Security-Type System In Figure A.1 we introduce SC, the security-type system for programs written in λ_2. The syntax is parameterized by a security policy specified in the form of a lattice structure on the set of labels $(L, \sqsubseteq)$. The types reflect those in λ_2 and include a new type constructor S_l for each label l. Typing judgements are of the form $\pi\ ;\ \Gamma \vdash^{\mathrm{SC}} t : A$ where: π is a *finite* set of labels drawn from L, i.e. $\pi \subseteq L$; Γ is an SC typing context; and A is an SC type. The component π is analogous to protection contexts from related work by Tse and S. Zdancewic (2004).

The set of labels π in the typing judgement represents the sensitivities of all the data on which the program may depend. In order to clarify the role of π, let us consider a program with a typing derivation indexed by the set of labels $\pi_1 \coloneqq \{\mathtt{H}\}$, $\pi_1\ ;\ \cdot \vdash^{\mathrm{SC}} p : A$. This program may depend on data labelled at types $\mathsf{S}_{\mathtt{L}}\ A$ and $\mathsf{S}_{\mathtt{H}}\ A$ by unsealing. If, instead, p is indexed by the set $\pi_2 \coloneqq \{\mathtt{L}\}$, i.e. $\pi_2\ ;\ \cdot \vdash^{\mathrm{SC}} p : A$, then the only terms that the program can depend on by unsealing are of type $\mathsf{S}_{\mathtt{L}}\ A$. This mechanism ensures that the flows of information are secure. It is useful to think that the labels that belong to π act as a kind of type-level *key* whose "possession" permits access to information at most as sensitive as the label itself.

The typing rules of the λ_2 fragment of SC, i.e. Rules Fun, App and If, are rather standard: they simply propagate the set of labels π to their premises. Observe that one has to explicitly unseal sensitive Booleans, i.e. of type $\mathsf{S}_l\ \mathsf{Bool}$, in order to branch on them using the Rule If. Rules Unseal and Seal are the most interesting since they enforce that information flows to the appropriate places. Rule Seal serves a double purpose: from premise to conclusion, it introduces terms of type $\mathsf{S}_l\ A$; and, from conclusion to premise, it extends the set of labels with the label l, i.e. $\pi \cup \{l\}$. The typing derivation above the premise can then unseal any term of type $\mathsf{S}_{l'}\ A$ such that its label l' can flow

$$\begin{array}{rl} \text{Sets of labels} & \pi \subseteq L \\ \text{Types} & A, B ::= \mathsf{Unit} \mid \mathsf{Bool} \mid A \Rightarrow B \mid \mathsf{S}_l\, A \\ \text{Typing contexts} & \Gamma ::= \cdot \mid \Gamma, x : A \end{array}$$

$\boxed{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} t : A}$

$$\textsc{Var}\ \frac{(x : A) \in \Gamma}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} x : A} \qquad \textsc{Fun}\ \frac{\pi \,;\, \Gamma, x : A \vdash^{\mathrm{SC}} t : B}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \lambda\, x.\, t : A \Rightarrow B}$$

$$\textsc{App}\ \frac{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} t : A \Rightarrow B \qquad \pi \,;\, \Gamma \vdash^{\mathrm{SC}} u : A}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mathsf{app}\, t\, u : B}$$

$$\textsc{Unit}\ \frac{}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mathsf{unit} : \mathsf{Unit}} \qquad \textsc{True}\ \frac{}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mathsf{true} : \mathsf{Bool}} \qquad \textsc{False}\ \frac{}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mathsf{false} : \mathsf{Bool}}$$

$$\textsc{If}\ \frac{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} t : \mathsf{Bool} \qquad \pi \,;\, \Gamma \vdash^{\mathrm{SC}} u_1 : A \qquad \pi \,;\, \Gamma \vdash^{\mathrm{SC}} u_2 : A}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mathsf{ifte}\, t\, u_1\, u_2 : A}$$

$$\textsc{Seal}\ \frac{\pi \cup \{l\} \,;\, \Gamma \vdash^{\mathrm{SC}} t : A}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mathsf{seal}_l\, t : \mathsf{S}_l\, A} \qquad \textsc{Unseal}\ \frac{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} t : \mathsf{S}_l\, A \qquad \exists l' \in \pi.\, l \sqsubseteq l'}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mathsf{unseal}_l\, t : A}$$

Figure A.1.: Types and intrinsically-typed terms of SC

to l. Rule Unseal allows unsealing a term with type $\mathsf{S}_l\, A$ if the set π in its conclusion contains at least a label l' such that $l \sqsubseteq l'$. Continuing with the intuition of labels in π as keys, a key $l' \in \pi$ can be used to unseal terms of type $\mathsf{S}_l\, A$ exactly when $l \sqsubseteq l'$.

In order to use SC as a security-type system for λ_2 programs, we define a family of erasure functions from SC types, typing contexts, and terms $\pi \,;\, \Gamma \vdash^{\mathrm{SC}} t : A$

to λ_2 types, typing contexts, and programs $\epsilon(\Gamma) \vdash \epsilon(t) : \epsilon(A)$:

$$\begin{array}{ll} \epsilon(\mathsf{Unit}) = \mathsf{Unit} & \epsilon(\lambda\, x.\, t) = \lambda\, x.\, \epsilon(t) \\ \epsilon(\mathsf{Bool}) = \mathsf{Bool} & \epsilon(\mathsf{app}\, t\, u) = \mathsf{app}\, \epsilon(t)\, \epsilon(u) \\ \epsilon(A \Rightarrow B) = \epsilon(A) \Rightarrow \epsilon(B) & \epsilon(\mathsf{seal}_l\, t) = \epsilon(t) \\ \epsilon(\mathsf{S}_l\, A) = \epsilon(A) & \epsilon(\mathsf{unseal}_l\, t) = \epsilon(t) \end{array}$$

$$\begin{array}{l} \epsilon(\cdot) = \cdot \\ \epsilon(\Gamma, x : A) = \epsilon(\Gamma), x : \epsilon(A) \end{array}$$

To clarify this point further, the noninterference property enforced by an SC term $\{\mathtt{L}\}\ ;\ sec : \mathsf{S}_{\mathtt{H}}\, \mathsf{Bool} \vdash^{\mathrm{SC}} t : \mathsf{Bool}$ on its underlying λ_2 program $sec : \mathsf{Bool} \vdash \epsilon(t) : \mathsf{Bool}$ is that for all $\cdot \vdash s_1, s_2 : \mathsf{Bool}$ both $\epsilon(t)[s_1/sec]$ and $\epsilon(t)[s_2/sec]$ terminate with the same Boolean. In this way, SC types and typing contexts play the role of security specifications, and SC terms of evidence that the underlying programs are secure, i.e. they satisfy the security specification. In contrast with SC, SC terms do not come equipped with an operational semantics, only their underlying λ_2 programs do. However, when convenient we identify SC terms with their erased λ_2 programs. To conclude, we observe that the security-type system is closed under the operational semantics of λ_2:

Lemma A.2.1. *Given a term $\pi\ ;\ \cdot \vdash^{SC} t : A$ such that $\epsilon(t) \to p$ then there exists a term $\pi\ ;\ \cdot \vdash^{SC} t' : A$ such that $\epsilon(t') = p$.*

A.3. Effectful Information-Flow Control

In this section, we present the main contribution of this paper: the observation that a single primitive distr is enough to enforce IFC in pure languages with effects. We study two extensions of the programming language and the security-type system from Section A.2 with printing and global store.

In these extensions, we treat effects *explicitly* in the style of Haskell's IO monad (Jones and Wadler 1993), Moggi's monadic metalanguage (Moggi 1991), or Katsumata's explicit subeffecting calculus (EFe) (Katsumata 2014, Section 5): the only programs that can perform effects are of type $\mathsf{Eff}_C\, a$ for some effect annotation C and type a, and sequencing of effects is made explicit through the primitive bind. Specifically, we consider printing and global store effects, Sections A.3.1 and A.3.2, respectively, as suitable representatives of the two kinds of effects that need to be secured:

Printing Effects. Printing on a channel can be observed externally to the program by the channel's observers. Observers can infer information

about the program's input from what is being printed to the channel. To secure printing effects one must ensure that the decision to print and what is being printed only depends on data less sensitive than the channel's observers.

Global Store Effects. Reading from the store cannot be observed directly. However, reading effects need to be secured because what is read may influence the program's subsequent behaviour. To secure reading effects one must ensure that what has been read is tracked as sensitive data.

In contrast to Haskell and the metalanguage, we use a graded monad (Katsumata 2014) whose effect annotation C tracks precisely to which channels a computation might print and which store locations a computation might access.

A.3.1. Printing Effects

In Figure A.2 we present the extension of λ_2 which allows programs to perform printing effects. We dub this language λ_2^{PRINT}. We assume that the set of printing channels Ch is fixed a priori, i.e. the channels are statically known. The set of types is extended with a new type for computations $\mathsf{Eff}_C\, a$ that is indexed by a set of channels $C \subseteq Ch$. A program of type $\mathsf{Eff}_C\, a$ when executed might *only* print to the channels that appear in C and return a result of type a.

Statics The typing rules of the standard monadic operations, return and bind, are as expected: in Rule Return, the computation does not perform any effects thus the type is indexed by the empty set of channels; and, in Rule Bind, the type is indexed by the union of the channels on which the computations $\Gamma \vdash t : \mathsf{Eff}_{C_1}\, a$ and $\Gamma \vdash u : a \Rightarrow \mathsf{Eff}_{C_2}\, b$ might print, that is, $C_1 \cup C_2$. We include subeffecting—casting from a smaller to a larger set of channels—as the term subeff in the language, see Rule Subeff. Printing is performed via a family of primitive operations, print_{ch}, one for each available channel $ch \in Ch$ (Rule Print). We assume, for simplicity, that only Boolean values can be printed. Further, observe that the resulting monadic type is indexed by the singleton set that only contains the channel on which the printing is performed, i.e. $\mathsf{Eff}_{\{ch\}}\, \mathsf{Unit}$.

Dynamics The operational semantics of λ_2^{PRINT} is defined as the combination of the small-step operational semantics of λ_2—see Appendix A.I for more details—and the small-step operational semantics of computations defined in Figure A.2. The semantics of effect-free terms, inherited from λ_2, $t \to u$,

$$
\begin{array}{rl}
\text{Sets of channels} & C, C_1, C_2 \subseteq Ch \\
\text{Outputs} & o, o_1, o_2 \in \text{List}\,(Ch \times 2) \\
\text{Types} & a, b ::= \ \ldots \mid \mathsf{Eff}_C\, a \\
\text{Typing contexts} & \Gamma ::= \ \ldots
\end{array}
$$

$\boxed{\Gamma \vdash t : a}$

$$\textsc{Return}\quad \frac{\Gamma \vdash t : a}{\Gamma \vdash \mathsf{return}\, t : \mathsf{Eff}_\emptyset\, a} \qquad \textsc{Bind}\quad \frac{\Gamma \vdash t : \mathsf{Eff}_{C_1} a \qquad \Gamma \vdash u : a \Rightarrow \mathsf{Eff}_{C_2} b}{\Gamma \vdash \mathsf{bind}\, t\, u : \mathsf{Eff}_{C_1 \cup C_2} b}$$

$$\textsc{Subeff}\quad \frac{\Gamma \vdash t : \mathsf{Eff}_{C_1} a \qquad C_1 \subseteq C_2}{\Gamma \vdash \mathsf{subeff}\, t : \mathsf{Eff}_{C_2}\, a} \qquad \textsc{Print}\quad \frac{\Gamma \vdash t : \mathsf{Bool}}{\Gamma \vdash \mathsf{print}_{ch}\, t : \mathsf{Eff}_{\{ch\}}\, \mathsf{Unit}}$$

$\boxed{t \rightsquigarrow u, o \text{ with } \cdot \vdash t : \mathsf{Eff}_C\, a \text{ and } \cdot \vdash u : \mathsf{Eff}_C\, a}$

$$\textsc{Bind}\quad \frac{t \rightsquigarrow t', o}{\mathsf{bind}\, t\, u \rightsquigarrow \mathsf{bind}\, t'\, u, o} \qquad \textsc{Bind-Subeff}\quad \frac{\text{value}\, t}{\mathsf{bind}\, (\mathsf{subeff}\, t)\, u \rightsquigarrow \mathsf{subeff}\, (\mathsf{bind}\, t\, u), \epsilon}$$

$$\textsc{Bind-Ret}\quad \frac{}{\mathsf{bind}\, (\mathsf{return}\, t)\, u \rightsquigarrow \mathsf{app}\, u\, t, \epsilon} \qquad \textsc{Subeff}\quad \frac{t \rightsquigarrow t', o}{\mathsf{subeff}\, t \rightsquigarrow \mathsf{subeff}\, t', o}$$

$$\textsc{Print}\quad \frac{t \to t'}{\mathsf{print}_{ch}\, t \rightsquigarrow \mathsf{print}_{ch}\, t', \epsilon} \qquad \textsc{Print-Val}\quad \frac{\text{value}\, b}{\mathsf{print}_{ch}\, b \rightsquigarrow \mathsf{return}\, \mathsf{unit}, [(ch, b)]} \qquad \textsc{EffectFree}\quad \frac{t \to t'}{t \rightsquigarrow t', \epsilon}$$

Figure A.2.: Types, well-typed terms and small-step semantics of λ_2^{PRINT} (omitting the unchanged rules of Appendix A.I)

treats monadic terms, such as $\mathsf{return}\, t$, as values even when their subterms are not values, e.g. $\mathsf{return}\,(\mathsf{app}\,(\lambda x.\, x)\,\mathsf{true}) \not\rightarrow$. The semantics of computations (alternatively, monadic semantics) is of the form $t \rightsquigarrow u, o$ and is interpreted as follows: program $\cdot \vdash t : \mathsf{Eff}_C\, a$ evaluates in one step to program $\cdot \vdash u : \mathsf{Eff}_C\, a$ and produces output o. The output is a list of pairs of channels and Boolean values $o \in \mathrm{List}\,(Ch \times \mathbb{2})$, and it represents the outputs of the program during execution.

We now briefly explain the semantics. Rule Bind reduces the left subterm of bind and executes its effects o. Once the left subterm is a value, i.e. $\mathsf{return}\, t$, Bind-Ret applies the rest of the computation u to the underlying value of type a, i.e. t. Applying the continuation u does not produce effects—recall that we are in a pure language—thus the step contains the empty output on the right, i.e. ϵ. Rule Print reduces the argument t of $\mathsf{print}_{ch}\, t$ until it is a value of type Bool, either true or false, and then Rule Print-Val prints the corresponding Boolean on the output channel ch. The output $[(ch, v)]$ is the singleton list with only one pair. Observe that these rules make $\mathsf{print}_{ch}\, t$ strict in its argument. Rule EffectFree serves to lift effect-free reductions to the level of computations. Since by definition effect-free reductions do not produce effects, the right hand side of the effect contains the empty output ϵ. To complete the picture, we denote by $t \rightsquigarrow^* u, o$ the reflexive-transitive closure of the monadic reduction relation. To combine effects, we use the monoid structure on $\mathrm{List}\,(Ch \times \mathbb{2})$.

Two Reduction Relations While it might seem unnecessary to define the semantics using the combination of a small-step relation of effect-free programs and a small-step relation of computations, it is a natural form of expressing the operational semantics of pure languages with effects (Wadler and Thiemann 2003). The effect-free relation evaluates programs that *cannot* perform effects, whilst the relation for computations evaluates programs which can, and computes those effects.

To better clarify this point, let us consider the execution on the following program, which we show in prettified syntax[5]:

$$
\begin{aligned}
&\mathit{prog}_5 : \mathsf{Eff}_{\{ch_1, ch_2\}}\, \mathsf{Unit} \\
&\mathit{prog}_5 = \mathsf{if}\ \mathsf{true}\ \mathsf{then} \\
&\qquad\qquad \mathsf{print}_{ch_1}\ \mathsf{false}\ \mathsf{else}\ \mathsf{return}\ \mathsf{unit} \\
&\qquad\qquad \gg\!= \lambda x.\, \mathsf{print}_{ch_2}\ \mathsf{true}
\end{aligned}
$$

The program evaluates as follows: 1. the effect-free semantics reduces the

[5]When possible we use Haskell-like syntax and leave the term subeff implicit.

term on the left of the bind, i.e. $\mathsf{if\,true\,then\,print}_{ch_1}\,\mathsf{false\,else\,return\,unit}$, to the value $\mathsf{print}_{ch_1}\,\mathsf{false}$ (Rule IF-TRUE through BIND) and this causes no effect; 2. the monadic semantics performs the effect, i.e. printing true on ch_1, (Rule PRINT-VAL); 3. the rest of the computation $\lambda\, x.\, \mathsf{print}_{ch_2}\,\mathsf{true}$ is applied to the resulting value, unit (Rule BIND-RET); 4. the effect-free semantics reduces the application (Rule BETA through EFFECTFREE) and finally; 5. the monadic semantics performs the effects of $\mathsf{print}_{ch_2}\,\mathsf{true}$.

To conclude the presentation of λ_2^{PRINT}, we enunciate the following lemma which states that the index C in the type of computations $\mathsf{Eff}_C\, a$ is a sound approximation of the set of channels where a program $\cdot \vdash t : \mathsf{Eff}_C\, a$ may print, i.e. t does not produce output in any channel not in C. In the security literature (e.g. (Volpano et al. 1996)) it is usually called *confinement*. Let us denote by $o|_{ch}$ the projection from o to the list consisting of all those pairs whose first component is the channel ch.

Lemma A.3.1 (Confinement for λ_2^{PRINT})**.** *For any λ_2^{PRINT} program of type $\cdot \vdash p : \mathsf{Eff}_C\, a$, λ_2^{PRINT} value $\cdot \vdash v : \mathsf{Eff}_C\, a$, and output o, if $p \rightsquigarrow^* v, o$ then $\forall ch \in Ch.\, ch \notin C \Rightarrow o|_{ch} = []$.*

Security-Type System After having defined the programming language, we are in position to turn our attention to the extension of SC that enforces IFC on λ_2^{PRINT}. We assume that the security policy specifies the sensitivity of each printing channel, i.e. the greatest lower bound of the sensitivities of all its observers, in the form of a function $\mathsf{label} \in Ch \to L$. In Figure A.3 we present the extension of SC that accommodates printing effects. We name it $\lambda_{\text{SC}}^{\text{PRINT}}$ hereafter. The types are the same as those in SC with the addition of a new type constructor Eff_C of computations. The typing rules for the λ_2^{PRINT} fragment of $\lambda_{\text{SC}}^{\text{PRINT}}$, i.e. Rules RETURN, BIND, SUBEFF and PRINT, simply propagate the set of labels π to their premises. Observe, again, that one has to explicitly unseal sensitive Booleans, i.e. of type $\mathsf{S}_l\,\mathsf{Bool}$, to apply the Rule PRINT. Before detailing Rule DISTR, we extend the family of erasure functions ϵ from $\lambda_{\text{SC}}^{\text{PRINT}}$-terms to λ_2^{PRINT} programs in the obvious way: i.e. the type former Eff_C erases to "itself" and, analogously to seal_l and unseal_l, distr is a no-op.

$$\begin{array}{ll} \epsilon(\dots) = \dots & \epsilon(\dots) = \dots \\ \epsilon(\mathsf{Eff}_C\, A) = \mathsf{Eff}_C\, \epsilon(A) & \epsilon(\mathsf{distr}\, t) = \epsilon(t) \end{array}$$

Rule DISTR introduces one of the novelties of our work; an enforcement mechanism that selectively permits to execute the effects of computations that depend on sensitive data. In $\lambda_{\text{SC}}^{\text{PRINT}}$, a term of type $\mathsf{S}_l\,(\mathsf{Eff}_C\, A)$ describes a

$$
\begin{array}{rl}
\text{Sets of channels} & C, C_1, C_2 \subseteq Ch \\
\text{Sensitivity of channels} & \mathsf{label} \in Ch \to L \\
\text{Sets of labels} & \pi \subseteq L \\
\text{Types} & A, B ::= \ \ldots \mid \mathsf{Eff}_C\, A \\
\text{Typing contexts} & \Gamma ::= \ \ldots
\end{array}
$$

$\boxed{\pi \ ; \ \Gamma \vdash^{\mathrm{SC}} t : A}$

$$\textsc{Return}\quad \frac{\pi \ ; \Gamma \vdash^{\mathrm{SC}} t : A}{\pi \ ; \Gamma \vdash^{\mathrm{SC}} \mathsf{return}\, t : \mathsf{Eff}_{\emptyset}\, A}$$

$$\textsc{Bind}\quad \frac{\pi \ ; \Gamma \vdash^{\mathrm{SC}} t : \mathsf{Eff}_{C_1}\, A \qquad \pi \ ; \Gamma \vdash^{\mathrm{SC}} u : A \Rightarrow \mathsf{Eff}_{C_2}\, B}{\pi \ ; \Gamma \vdash^{\mathrm{SC}} \mathsf{bind}\, t\, u : \mathsf{Eff}_{C_1 \cup C_2}\, B}$$

$$\textsc{Subeff}\quad \frac{\pi \ ; \Gamma \vdash^{\mathrm{SC}} t : \mathsf{Eff}_{C_1}\, A \qquad C_1 \subseteq C_2}{\pi \ ; \Gamma \vdash^{\mathrm{SC}} \mathsf{subeff}\, t : \mathsf{Eff}_{C_2}\, A}$$

$$\textsc{Print}\quad \frac{\pi \ ; \Gamma \vdash^{\mathrm{SC}} t : \mathsf{Bool}}{\pi \ ; \Gamma \vdash^{\mathrm{SC}} \mathsf{print}_{ch}\, t : \mathsf{Eff}_{\{ch\}}\, \mathsf{Unit}}$$

$$\textsc{Distr}\quad \frac{\pi \ ; \Gamma \vdash^{\mathrm{SC}} t : \mathsf{S}_l\, (\mathsf{Eff}_C\, A)}{\pi \ ; \Gamma \vdash^{\mathrm{SC}} \mathsf{distr}\, t : \mathsf{Eff}_C\, (\mathsf{S}_l\, A)}\ (\forall ch \in C.\ l \sqsubseteq \mathrm{label}(ch))$$

Figure A.3.: Types and intrinsically-typed terms of $\lambda_{\mathrm{SC}}^{\mathrm{PRINT}}$ (omitting the unchanged rules of Figure A.1)

computation that might *only* print on the channels in C and what is printed and the decision to print potentially depends on data of sensitivity l. Then, it is natural to ask, when it is secure to execute the effects of the inner computation of type $\mathsf{Eff}_C\, A$? Clearly, whenever the sensitivities of the computation's effects, i.e. the channels in the set C, are as high as the sensitivity of the data used to decide to perform the effects, i.e. sensitivity l. The side-condition of the rule exactly captures this condition: $\forall ch \in C.\ l \sqsubseteq \mathrm{label}(ch)$.

To illustrate Distr in action, let us consider the following term in $\lambda_{\mathrm{SC}}^{\mathrm{PRINT}}$ that prints Alice's sensitive input to the H-sensitive channel:

$$
\begin{aligned}
&\mathit{prog}_6 : \mathsf{S}_{\mathtt{A}}\,\mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{H}}\}}\,\mathsf{Unit}\\
&\mathit{prog}_6 = \lambda\, sb.\, \mathsf{distr}\,(\mathsf{seal}_{\mathtt{A}}\,(\mathsf{print}_{\mathtt{Ch}_{\mathtt{H}}}\,(\mathsf{unseal}_{\mathtt{A}}\, sb))) \gg\!= \lambda\, x.\, \mathsf{return}\,\mathsf{unit}
\end{aligned}
$$

The primitive distr permits to execute the effects of the computation inside the term $\mathsf{seal}_{\mathtt{A}}\,(\mathsf{print}_{\mathtt{Ch}_{\mathtt{H}}}\,(\mathsf{unseal}_{\mathtt{A}}\, sb))$. The term distr protects the return type of the computation at the same sensitivity as the premise's type $\mathsf{S}_l\, A$. This requirement is necessary to enforce IFC because the trivial computation that performs no effects and just returns has access to sensitive data, as exemplified by the following program:

$$
\begin{aligned}
&\mathit{prog}_7 : \mathsf{S}_{\mathtt{A}}\,\mathsf{Bool} \Rightarrow \mathsf{S}_{\mathtt{B}}\,\mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{H}}\}}\,(\mathsf{S}_{\mathtt{H}}\,\mathsf{Bool})\\
&\mathit{prog}_7 =\\
&\quad \lambda\, sb_1\, sb_2.\, \mathsf{distr}\,(\mathsf{seal}_{\mathtt{H}}\,(\mathsf{print}_{\mathtt{Ch}_{\mathtt{H}}}\,(\mathsf{unseal}_{\mathtt{A}}\, sb_1)) \gg\!= \lambda\, x.\, \mathsf{return}\,(\mathsf{unseal}_{\mathtt{B}}\, sb_2))
\end{aligned}
$$

A.3.2. Global Store Effects

We turn our attention to global store effects which combines printing effects from the previous section with reading from locations in the store. Following the same steps, in Figure A.4 we present the extension of λ_2 (Appendix A.I) in which programs have access to a global store and can read from and write to it. We name this language $\lambda_2^{\mathrm{STORE}}$. Our development rests on two assumptions: (1) the set of locations in the store *Loc* is fixed during execution, and (2) only terms of ground type, i.e. Bool and Unit, can be stored. This helps simplify the presentation of the language and the security-type system, and, as we will show in the next section, the construction of the logical relation from which noninterference follows. At the end of this section, however, we briefly discuss how to lift these assumptions.

$\lambda_2^{\mathrm{STORE}}$ extends λ_2 with a type of computations $\mathsf{Eff}_S\, a$, which is indexed by a set of store locations $S \subseteq \mathit{Loc}$, and a type of references $\mathsf{Ref}_s\, r$, which is indexed by store locations $s \in S$. A program of type $\mathsf{Eff}_S\, a$ when executed might *only* write to the references mentioned in the set S and finally return a result of

type a. A term of type $\mathsf{Ref}_s\, r$ is a reference in the store s that contains terms of ground type r. References permit both "printing" effects via writing—like channels in λ_2^{PRINT}—but also reading effects. Note that the annotation S in the type of computations Eff_S *does not* mention the locations on the store from which the program might read. This asymmetry stems from how the execution of programs performing these effects interact with their environment: writing alters the store whilst reading does not.

Statics Typing judgements are of the form $\Sigma, \Gamma \vdash t : a$ where the store typing Σ determines the shape of the store, i.e. what types it contains and in what locations. The rest of components are like those in λ_2. The typing rules of the monadic operations return, bind and subeff follow the same pattern as in λ_2^{PRINT} (Figure A.2), thus we do not discuss them any further. The term ref_s (Rule Ref) is the runtime representation of references. Reading and writing is achieved via primitives read and write (Rules Read and Write respectively). Observe that in Read, the type of the computation $\mathsf{Eff}_\emptyset\, r$ in the conclusion of the rule is indexed by the empty set of locations, while in Write is indexed by the singleton set $\{s\}$.

Dynamics The operational semantics of the language is defined as in λ_2^{PRINT}; a combination of the small-step semantics for λ_2 that treats effectful primitives as values, and a small-step semantics for computations. Given a store typing Σ, a store θ is a function from locations to typed terms according to Σ. Since the language considers a fixed-size store, we use the notation θ instead of $\theta(\Sigma)$. The semantics of computations is of the form $\theta_1, t \rightsquigarrow \theta_2, u$, and is interpreted as: program $\Sigma, \cdot \vdash t : \mathsf{Eff}_S\, a$ paired with store θ_1 evaluates in one step to program $\Sigma, \cdot \vdash u : \mathsf{Eff}_S\, a$ and store θ_2.

We now explain the semantics. Rule Read reduces the argument of read and it does not modify the store. When the argument is a store location, i.e. ref_s, Read-Ref retrieves the term t from the store, i.e. $\theta(s) = t$. The rules for writing Rules Write and Write-Ref first reduce the left subterm of $\mathsf{write}\, u\, t$ to a store location and then write the right subterm t on the store. Different from λ_2^{PRINT}, we permit to write any term on the store, not only values.

To briefly illustrate λ_2^{STORE}, consider the following program:

$$
\begin{aligned}
&\mathit{prog}_{\mathcal{S}} : \mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{s'\}}\, \mathsf{Unit} \\
&\mathit{prog}_{\mathcal{S}}\, b = \mathsf{if}\ b\ \mathsf{then}\ (\mathsf{read}\, s \gg\!= \lambda x.\, \mathsf{write}\, s'\, x)\ \mathsf{else}\ \mathsf{return}\ \mathsf{unit}
\end{aligned}
$$

Based on the Boolean input, $\mathit{prog}_{\mathcal{S}}$ copies the contents of the store location s to s'. Observe that the type only mentions the location s' in its index.

$$
\begin{array}{rl}
\text{Store locations} & s \in \mathit{Loc} \\
\text{Sets of s. locations} & S, S_1, S_2 \subseteq \mathit{Loc} \\
\text{Ground types} & r ::= \mathsf{Bool} \mid \mathsf{Unit} \\
\text{Types} & a, b ::= r \mid a \Rightarrow b \mid \mathsf{Eff}_S\, a \mid \mathsf{Ref}_s\, r \\
\text{Typing contexts} & \Gamma ::= \ldots \\
\text{Store typings} & \Sigma \in \mathit{Loc} \to \text{Ground types}
\end{array}
$$

$\boxed{\Sigma, \Gamma \vdash t : a}$

$$
\textsc{Ref}\ \frac{\Sigma(s) = r}{\Sigma, \Gamma \vdash \mathsf{ref}_s : \mathsf{Ref}_s\, r} \qquad \textsc{Read}\ \frac{\Sigma, \Gamma \vdash t : \mathsf{Ref}_s\, r}{\Sigma, \Gamma \vdash \mathsf{read}\, t : \mathsf{Eff}_\emptyset\, r}
$$

$$
\textsc{Write}\ \frac{\Sigma, \Gamma \vdash t : \mathsf{Ref}_s\, r \qquad \Sigma, \Gamma \vdash u : r}{\Sigma, \Gamma \vdash \mathsf{write}\, t\, u : \mathsf{Eff}_{\{s\}}\, \mathsf{Unit}}
$$

Stores $\quad \theta, \theta_1, \theta_2 \quad \in (\Sigma : \text{Store typing}) \to (s : \mathit{Loc}) \to \Sigma\ , \cdot \vdash t : \Sigma(s)$

$\boxed{\theta_1(\Sigma), t \rightsquigarrow \theta_2(\Sigma), u \text{ with } \Sigma, \cdot \vdash t : \mathsf{Eff}_S\, a \text{ and } \Sigma, \cdot \vdash u : \mathsf{Eff}_S\, a}$

$$
\textsc{Read}\ \frac{t \to u}{\theta, \mathsf{read}\, t \rightsquigarrow \theta, \mathsf{read}\, u} \qquad \textsc{Read-Ref}\ \frac{\theta(s) = t}{\theta, \mathsf{read}\, \mathsf{ref}_s \rightsquigarrow \theta, \mathsf{return}\, t}
$$

$$
\textsc{Write}\ \frac{t \to t'}{\theta, \mathsf{write}\, t\, u \rightsquigarrow \theta, \mathsf{write}\, t'\, u} \qquad \textsc{Write-Ref}\ \frac{\theta_2 = \theta_1[s \mapsto t]}{\theta_1, \mathsf{write}\, \mathsf{ref}_s\, t \rightsquigarrow \theta_2, \mathsf{return}\, \mathsf{unit}}
$$

Figure A.4.: Types, well-typed terms and small-step semantics of λ_2^{STORE} (omitting the unchanged rules of Appendix A.I and Figure A.2)

We conclude the explanation of λ_2^{STORE} with a confinement lemma similar to that of λ_2^{PRINT} (Lemma A.3.1):

Lemma A.3.2 (Confinement for λ_2^{STORE})**.** *For any λ_2^{STORE} program $\Sigma, \cdot \vdash f : \mathsf{Eff}_S\, a$, λ_2^{STORE} value $\Sigma, \cdot \vdash v : \mathsf{Eff}_S\, a$, and stores $\theta_2, \theta_2 : \Sigma$, if $\theta_1, f \rightsquigarrow^* \theta_2, v$ then $\forall s \in Loc.\, s \notin S \Rightarrow \theta_1(s) = \theta_2(s)$.*

Security-Type System Now we explain the IFC enforcement mechanism $\lambda_{\text{SC}}^{\text{STORE}}$ (Figure A.5). As in $\lambda_{\text{SC}}^{\text{PRINT}}$ (Figure A.3) we assume that the security policy specifies for each store location its sensitivity as a function label $\in Loc \to L$. The typing rules for the monadic primitives are analogous to $\lambda_{\text{SC}}^{\text{PRINT}}$ thus we have omitted them. The rule for references is straightforward (Rule Ref). More interesting is the typing rule for reading from the store (Rule Read). In the conclusion of the rule, the return type of the computation is the SC type for sensitive data $\mathsf{S}_l\, R$. The sensitivity of the location is l, i.e. $\text{label}(s) = l$ in the side-condition of the rule, thus to protect the flow of information is necessary to wrap also the return type.

The type of read diverges from usual presentations of IFC libraries (e.g. MAC (Russo 2015) and HLIO (Buiras et al. 2015) with the exception of SLIO (Rajani and Garg 2020)) in that the result of reading from the store is wrapped in the type constructor S_l. These libraries incorporate the data into their monad of computations, which keeps track of the sensitivities of the observed values.

We conclude the section with a concrete example of $\lambda_{\text{SC}}^{\text{STORE}}$ that shows that $prog_8$ is secure with respect to the following specification: $\text{label}(s) = \mathtt{A}$, $\text{label}(s') = \mathtt{H}$ and the sensitivity of the Boolean argument is $\mathtt{H}$, i.e. $sb : \mathsf{S}_{\mathtt{H}}\, \mathsf{Bool}$:

$$
\begin{aligned}
&prog'_8 : \mathsf{S}_{\mathtt{H}}\, \mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{s'\}}\, (\mathsf{S}_{\mathtt{H}}\, \mathsf{Unit}) \\
&prog'_8\ sb = \mathsf{distr}(\mathsf{seal}_{\mathtt{H}}(\mathsf{if}\ \mathsf{unseal}_{\mathtt{A}}\ sb \\
&\qquad\qquad\qquad\qquad \mathsf{then}\ (\mathsf{read}\ s \gg\!= \lambda x.\, \mathsf{write}\ s'\ (\mathsf{unseal}_{\mathtt{H}}\ x)) \\
&\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{return}\ \mathsf{unit}))
\end{aligned}
$$

The above program exemplifies how $\lambda_{\text{SC}}^{\text{STORE}}$ enforces that flows from the store to the program and back to the store are secure. Note that $\epsilon(prog'_8) = prog_8$.

A.3.3. Other Effects, Combination of Effects

To conclude we note that the previous sections show how to treat, from an IFC perspective, reading and writing effects in a general sense. With such a development, our approach could be used to not only encode secure reading/writing

$$
\begin{array}{rr}
\text{Store locations} & s \in \mathit{Loc} \\
\text{Sets of s. locations} & S, S_1, S_2 \subseteq \mathit{Loc} \\
\text{Sensitivity of loc.} & \mathrm{label} \in \mathit{Loc} \to L \\
\text{Sets of labels} & \pi \subseteq L \\
\text{Ground types} & R ::= \ \mathsf{Bool} \mid \mathsf{Unit} \\
\text{Types} & A, B ::= \ R \mid A \Rightarrow B \mid \mathsf{Eff}_S\, A \mid \mathsf{Ref}_s\, R \\
\text{Typing contexts} & \Gamma ::= \ \ldots \\
\text{Store typings} & \Sigma \in \mathit{Loc} \to \text{Ground type}
\end{array}
$$

$\boxed{\pi \,;\, \Sigma, \Gamma \vdash^{\mathrm{SC}} t : A}$

$$
\textsc{Ref}\ \frac{\Sigma(s) = R}{\pi \,;\, \Sigma, \Gamma \vdash^{\mathrm{SC}} \mathsf{ref}_s : \mathsf{Ref}_s\, R} \qquad \textsc{Read}\ \frac{\pi \,;\, \Sigma, \Gamma \vdash^{\mathrm{SC}} t : \mathsf{Ref}_s\, R}{\pi \,;\, \Sigma, \Gamma \vdash^{\mathrm{SC}} \mathsf{read}\, t : \mathsf{Eff}_\emptyset\, (\mathsf{S}_l\, R)}\ (\mathrm{label}(s) = l)
$$

$$
\textsc{Write}\ \frac{\pi \,;\, \Sigma, \Gamma \vdash^{\mathrm{SC}} t : \mathsf{Ref}_s\, R \qquad \pi \,;\, \Sigma, \Gamma \vdash^{\mathrm{SC}} u : R}{\pi \,;\, \Sigma, \Gamma \vdash^{\mathrm{SC}} \mathsf{write}\, t\, u : \mathsf{Eff}_{\{s\}}\, \mathsf{Unit}}
$$

$$
\textsc{Distr}\ \frac{\pi \,;\, \Sigma, \Gamma \vdash^{\mathrm{SC}} t : \mathsf{S}_l\, (\mathsf{Eff}_S\, A)}{\pi \,;\, \Sigma, \Gamma \vdash^{\mathrm{SC}} \mathsf{distr}\, t : \mathsf{Eff}_S\, (\mathsf{S}_l\, A)}\ (\forall s \in S.\ l \sqsubseteq \mathrm{label}(s))
$$

Figure A.5.: Types and intrinsically-typed terms of $\lambda_{\mathrm{SC}}^{\mathrm{STORE}}$ (omitting the unchanged rules of Figure A.1)

of references but also effects involving files and network communications. However, our framework can be adapted to deal with other kinds of effects, e.g. exceptions, or combinations thereof by selecting what effects the graded monad has to track, e.g. what exceptions are thrown, and when it it secure to extract a computation from a sealed value.

A.4. Security Guarantees

In this section, we prove that the IFC mechanisms SC, $\lambda_{\text{SC}}^{\text{PRINT}}$ and $\lambda_{\text{SC}}^{\text{STORE}}$ can be used to enforce noninterference for the programming languages λ_2, λ_2^{PRINT} and λ_2^{STORE} presented in Sections A.2, A.3.1 and A.3.2, respectively. Our noninterference proofs employ the technique of logical relations (LRs). For each language, we construct a LR parameterized by the sensitivity of the attacker `Atk` and the security types of SC. In the effect-free setting, the LR interprets each SC type A as a binary relation over λ_2 programs of the erased type $\epsilon(A)$. The relation captures the idea of indistinguishable programs: if the type is *public* enough, e.g. $\mathsf{S}_{\mathtt{L}}\,\mathsf{Bool}$ and `L` flows to `Atk`, then two programs are related when they evaluate to the same value. Noninterference follows as a corollary of the so called fundamental theorem of the LR.

In this work we study languages without recursion where all programs terminate[6]. It is immediate, however, to extend these languages with recursive function definitions, which would allow writing nonterminating programs, and the noninterference proofs to appropriate variants of termination-insensitive noninterference using step-indexed LRs (A. J. Ahmed 2006)—for example following Gregersen et al. (2021). We note that, in fact, the security-type systems for λ_2^{PRINT} and λ_2^{STORE} extended with recursion enforce a stronger notion of security, namely progress-insensitive noninterference (PINI). In order to prove this property the (step-indexed) LRs require nontrivial generalizations that can deal with partial reduction sequences with observable effects. We declare this line of research as future work.

Noninterference (NI) states that for any two runs of a program with different secret inputs its public outputs agree, and thus the attacker cannot infer the contents of sensitive data. In the effect-free setting, the inputs to a program are its arguments, and the output is its return value. In the effectful setting, what we need to consider as inputs and outputs changes: in λ_2^{STORE}, for instance, the store must be considered an additional input to the program.

[6]This follows straightforwardly using a Tait-style reducibility predicate.

Definition A.4.1 (NI for λ_2)**.** A program $sec : \mathsf{Bool} \vdash p : \mathsf{Bool}$ satisfies NI if for any two terms $\cdot \vdash s_1, s_2 : \mathsf{Bool}$, and any two values $\cdot \vdash v_1, v_2 : \mathsf{Bool}$, such that $p[s_1/sec] \to^* v_1$ and $p[s_2/sec] \to^* v_2$ it is the case that $v_1 = v_2$.

In the definition $v_1 = v_2$ denotes that v_1 and v_2 are syntactically equal values.

SC enforces NI:

Theorem A.4.1. *Given any SC term* $\{\mathtt{Atk}\}\ ;\ sec : \mathsf{S}_{\mathtt{H}}\,\mathsf{Bool} \vdash^{SC} t : \mathsf{S}_{\mathtt{Atk}}\,\mathsf{Bool}$ *where* $\mathtt{H} \not\sqsubseteq \mathtt{Atk}$*, the erased program* $sec : \mathsf{Bool} \vdash \epsilon(t) : \mathsf{Bool}$ *satisfies NI.*

In practice, we are concerned that information does not flow from H-protected data to the attacker with sensitivity Atk. Thus, to show that a program $sec : \mathsf{Bool} \vdash p : \mathsf{Bool}$ does not leak information from H to Atk, it is enough to find an SC term $\{\mathtt{Atk}\}\ ;\ sec : \mathsf{S}_{\mathtt{H}}\,\mathsf{Bool} \vdash^{\mathrm{SC}} t : \mathsf{S}_{\mathtt{Atk}}\,\mathsf{Bool}$ such that $p = \epsilon(t)$.

Logical Relation In order to prove that SC enforces NI (Theorem A.4.1), we construct a LR parameterized by the attacker's sensitivity Atk and the SC types—see Figure A.6. The proof, as we will show, then falls out as a consequence of the fundamental theorem of the LR.

At each SC type the LR defines what the attacker can observe about pairs of λ_2 programs of erased type—alternatively, the same program with different secrets. We split the definition depending on whether programs are evaluated, i.e. they are already values, $\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![-]\!]$ and $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![-]\!]$, respectively. We briefly explain these definitions. At SC type Bool, for instance, see $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![\mathsf{Bool}]\!]$ and $\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![\mathsf{Bool}]\!]$, the LR states that if the programs terminate then the attacker can observe if they return equal values. At higher types, i.e. $A \Rightarrow B$, two functions are related if whenever they reduce to a value, see $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![A \Rightarrow B]\!]$, they map related inputs $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![A]\!](u_1, u_2)$ to related outputs $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![A]\!](\mathsf{app}\,t_1\,u_1, \mathsf{app}\,t_2\,u_2)$, see $\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![A \Rightarrow B]\!]$. Lastly, at type $\mathsf{S}_l\,A$, see $\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![\mathsf{S}_l\,A]\!]$, the LR compares the sensitivity of the attacker with the label l, and in case it is less sensitive, i.e. $l \sqsubseteq \mathtt{Atk}$, the programs have to be related at SC type A. If the label l is more sensitive than the attacker's label, i.e. $l \not\sqsubseteq \mathtt{Atk}$ then the programs do not need to be related.

Definitions $\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![-]\!]$ and $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![-]\!]$ work on closed terms, however, in order to prove the fundamental theorem we have to lift them to closed substitutions and open terms. A substitution assigns to each type a in a typing context Γ a closed term of that type, i.e. $\cdot \vdash t : a$. We denote substitutions by γ and use $\gamma : \cdot \vdash \Gamma$ to mean that γ is in the set of substitutions over Γ. We define the LR for substitutions, $\mathcal{R}_{\mathcal{S}}^{\mathtt{Atk}}[\![-]\!]$, by induction on SC typing contexts. At the empty context $\cdot$ the empty substitutions (ϵ, ϵ) are trivially related—denoted by $\top$. Two nonempty substitutions (γ_1, t_1) and (γ_2, t_2) are related whenever they

$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![-]\!] \in (A : \text{SC type}) \to (t_1 : \cdot \vdash \epsilon(A)) \to (t_2 : \cdot \vdash \epsilon(A)) \to \text{Set}$$
$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![\mathsf{Unit}]\!](t_1, t_2) :\Leftrightarrow t_1 = t_2$$
$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![\mathsf{Bool}]\!](t_1, t_2) :\Leftrightarrow t_1 = t_2$$
$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![A \Rightarrow B]\!](t_1, t_2) :\Leftrightarrow \forall(u_1, u_2 : \cdot \vdash \epsilon(A)).$$
$$\quad \mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![A]\!](u_1, u_2) \Rightarrow \mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![B]\!](\mathsf{app}\, t_1\, u_1, \mathsf{app}\, t_2\, u_2)$$
$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![\mathsf{S}_l\, A]\!](t_1, t_2) :\Leftrightarrow l \sqsubseteq \mathtt{Atk} \Rightarrow \mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![A]\!](t_1, t_2)$$

$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![A]\!](t_1, t_2) :\Leftrightarrow \forall(u_1, u_2 : \cdot \vdash \epsilon(A)).\, t_1 \to^* u_1 \wedge t_2 \to^* u_2 \Rightarrow \mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![A]\!](u_1, u_2)$$

$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{S}}[\![\cdot]\!](\epsilon, \epsilon) :\Leftrightarrow \top$$
$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{S}}[\![\Gamma, x : A]\!]((\gamma_1, t_1), (\gamma_2, t_2)) :\Leftrightarrow \mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![A]\!](t_1, t_2) \wedge \mathcal{R}^{\mathtt{Atk}}_{\mathcal{S}}[\![\Gamma]\!](\gamma_1, \gamma_2)$$

$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{T}}[\![(\Gamma, A)]\!](t_1, t_2) :\Leftrightarrow \forall(\gamma_1, \gamma_2 : \cdot \vdash \epsilon(\Gamma)).$$
$$\quad \mathcal{R}^{\mathtt{Atk}}_{\mathcal{S}}[\![\Gamma]\!](\gamma_1, \gamma_2) \Rightarrow \mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![A]\!](t_1[\gamma_1], t_2[\gamma_2])$$

Figure A.6.: Logical relation for λ_2-SC

are pointwise related, i.e. $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![A]\!](t_1, t_2)$ and $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{S}}[\![\Gamma]\!](\gamma_1, \gamma_2)$. The LR for open terms, written $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{T}}[\![(\Gamma, A)]\!]$, is indexed by a pair consisting of an SC typing context Γ and an SC type A. It states that two λ_2 terms are related if for any two related closing substitutions the substituted terms are related at type A.

The fundamental theorem of the LR states that the underlying program of an SC term is related to itself. Formally:

Theorem A.4.2 (Fundamental Theorem of the LR for λ_2-SC)**.** *For any attacker with sensitivity* $\mathtt{Atk}$*, and SC term* $\{\mathtt{Atk}\}\ ;\ \Gamma \vdash^{SC} t : A$*, it is the case that* $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{T}}[\![(\Gamma, A)]\!](\epsilon(t), \epsilon(t))$*.*

Proof. By induction on the typing derivation. □

NI (Theorem A.4.1) follows as a corollary of the fundamental theorem:

Proof. Assume two Boolean secrets $\cdot \vdash s_1, s_2 : \mathsf{Bool}$. Since $\mathtt{H} \not\sqsubseteq \mathtt{Atk}$, the secrets are related, i.e. $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![\mathsf{S}_{\mathtt{H}}\, \mathsf{Bool}]\!](s_1, s_2)$, and thus the two substitutions $\gamma_1 = \{sec \mapsto s_1\}$ and $\gamma_2 = \{sec \mapsto s_2\}$ are related. By the fundamental theorem, the term $\epsilon(t)$ is related to itself, i.e. $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![\mathsf{S}_{\mathtt{Atk}}\, \mathsf{Bool}]\!](\epsilon(t)[\gamma_1], \epsilon(t)[\gamma_2])$. Unfolding

$$\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![\ldots]\!](t_1,t_2) :\Leftrightarrow \ldots$$
$$\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![\mathsf{Eff}_C\, A]\!](t_1,t_2) :\Leftrightarrow \forall(u_1,u_2 : \cdot \vdash \epsilon(A)), o_1, o_2.\, t_1 \rightsquigarrow^* \mathsf{return}(u_1), o_1 \\ \wedge\, t_2 \rightsquigarrow^* \mathsf{return}(u_2), o_2 \Rightarrow \mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![A]\!](u_1,u_2) \wedge o_1 =_{C_{\mathtt{Atk}}} o_2$$

Figure A.7.: Logical relation for λ_2^{PRINT}-$\lambda_{\text{SC}}^{\text{PRINT}}$ (omitting the unchanged clauses of Figure A.6)

the definitions of we obtain that $\forall(v_1, v_2 : \cdot \vdash \mathsf{Bool}).\, \epsilon(t)[\gamma_1] \to^* v_1 \wedge \epsilon(t)[\gamma_2] \to^* v_2 \Rightarrow v_1 = v_2$. Since $\epsilon(t)[\gamma_1]$ and $\epsilon(t)[\gamma_2]$ terminate we obtain. □

A.4.1. Noninterference for Printing Effects

In order to formalize noninterference for λ_2^{PRINT} we look at effectful programs that might print. For that, we first define indistinguishability of outputs with respect to a subset of channels:

Definition A.4.2 (Output indistinguishability)**.** Let $C \subseteq Ch$. Two outputs o_1 and o_2 are C-indistinguishable, denoted by $o_1 =_C o_2$, if the two outputs agree in C, i.e. $o_1|_C = o_2|_C$.

We define NI for λ_2^{PRINT} for programs from Bool to $\mathsf{Eff}_C\,\mathsf{Unit}$, i.e. programs that depending on a Boolean produce output in an arbitrary set of channels C:

Definition A.4.3 (NI for λ_2^{PRINT})**.** A program $sec : \mathsf{Bool} \vdash p : \mathsf{Eff}_C\,\mathsf{Unit}$ satisfies NI *with respect to* $C' \subseteq C$, if for any two terms $\cdot \vdash s_1, s_2 : \mathsf{Bool}$, any two values $\cdot \vdash v_1, v_2 : \mathsf{Eff}_C\,\mathsf{Unit}$, and any two outputs o_1 and o_2, such that $p[s_1/sec] \rightsquigarrow^* v_1, o_1$ and $p[s_2/sec] \rightsquigarrow^* v_2, o_2$, then $o_1 =_{C'} o_2$.

NI is parameterized by a subset of the channels where the outputs have to agree. We will instantiate C' with the set of channels observable by the attacker.

$\lambda_{\text{SC}}^{\text{PRINT}}$ can be used to enforce NI on λ_2^{PRINT} programs:

Theorem A.4.3. *Given any* λ_{SC}^{PRINT} *term* $\{\mathtt{Atk}\}\ ;\ sec : \mathsf{S}_{\mathtt{H}}\,\mathsf{Bool} \vdash^{SC} t : \mathsf{Eff}_C\,\mathsf{Unit}$ *where* $\mathtt{H} \not\sqsubseteq \mathtt{Atk}$ *the erased program* $sec : \mathsf{Bool} \vdash \epsilon(t) : \mathsf{Eff}_C\,\mathsf{Unit}$ *satisfies NI* with respect to $C_{\mathtt{Atk}}$ *where* $C_{\mathtt{Atk}} := \{ch \mid ch \in C, \text{label}(ch) \sqsubseteq \mathtt{Atk}\}$.

Logical Relation The LR for λ_2^{PRINT} (Figure A.7) is very similar to that of λ_2 (Figure A.6) so we skip over the commonalities and directly discuss its definition at the type of computations. The LR relates two computations

$\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![\mathsf{Eff}_C\, A]\!](t_1, t_2)$ if whenever they terminate, i.e. $t_1 \rightsquigarrow^* \mathsf{return}\, u_1, o_1$ and $t_2 \rightsquigarrow^* \mathsf{return}\, u_2, o_2$, the resulting terms are related, i.e. $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![\tau]\!](u_1, u_2)$, and the outputs are indistinguishable by the attacker, i.e. $o_1 =_{C_{\mathtt{Atk}}} o_2$. The fundamental theorem of the LR states that erased terms are related to themselves:

Theorem A.4.4 (Fundamental Theorem of the LR for λ_2^{PRINT}-$\lambda_{\text{SC}}^{\text{PRINT}}$)**.** *For any attacker with sensitivity* $\mathtt{Atk}$*, and* λ_{SC}^{PRINT} *term* $\{\mathtt{Atk}\}\,; \Gamma \vdash^{SC} t : A$*, it is the case that* $\mathcal{R}_{\mathcal{T}}^{\mathtt{Atk}}[\![(\Gamma, A)]\!](\epsilon(t), \epsilon(t))$*.*

Proof. By induction on the typing derivation with use of the Lemma A.3.1. □

The proof that $\lambda_{\text{SC}}^{\text{PRINT}}$ enforces NI (Theorem A.4.3) requires the following Lemma.

Lemma A.4.1. *If* $p \rightsquigarrow^* v, o$*, then there exists a* $\rightarrow$*-value* p' *such that* $p \rightarrow^* p'$ *and* $p' \rightsquigarrow^* v, o$*.*

NI follows as a corollary of the fundamental theorem.

Proof. Let us assume two secret Booleans $\cdot \vdash s_1, s_2 : \mathsf{Bool}$. Since $\mathtt{H} \not\sqsubseteq \mathtt{Atk}$, the secrets are related, i.e. $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![\mathsf{S}_{\mathtt{H}}\, \mathsf{Bool}]\!](s_1, s_2)$, and thus the substitutions $\gamma_1 = \{sec \mapsto s_1\}$ and $\gamma_2 = \{sec \mapsto s_2\}$ are related. By the fundamental theorem, the term $\epsilon(t)$ is related to itself $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![\mathsf{Eff}_C\, \mathsf{Unit}]\!](\epsilon(t)[\gamma_1], \epsilon(t)[\gamma_2])$.

By assumption $\epsilon(t)[s_1/sec] \rightsquigarrow^* v_1, o_1$ and $\epsilon(t)[s_2/sec] \rightsquigarrow^* v_2', o_2$, and by Lemma A.4.1, there are two intermediate programs t_1' and t_2' such that: $\epsilon(t)[s_1/sec] \rightarrow^* t_1'$ and $t_1' \rightsquigarrow^* v_1, o_1$; and $\epsilon(t)[s_2/sec] \rightarrow^* t_2'$ and $t_2' \rightsquigarrow^* v_2, o_2$. We apply $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![\mathsf{Eff}_C\, \mathsf{Unit}]\!](\epsilon(t)[s_1/sec], \epsilon(t)[s_2/sec])$ to the two effect-free reductions which gives us that $\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![\mathsf{Eff}_C\, \mathsf{Unit}]\!](t_1', t_2')$. We apply this to the monadic reductions and obtain that $o_1 =_{C_{\mathtt{Atk}}} o_2$. □

A.4.2. Noninterference for Global Store Effects

We formalize noninterference for λ_2^{STORE} by looking at effectful programs which receive a store as input and produce a store as output. The contents of the store are possibly unevaluated λ_2^{STORE} programs of ground type (see Figure A.4). In order to compare stores, we define an indistinguishability relation for programs of ground type:

$$\mathcal{R}_{\mathcal{G}}[\![-]\!] : (r : \lambda_2^{\text{STORE}} \text{ ground type}) \to (t_1 : \cdot \vdash r) \to (t_2 : \cdot \vdash r) \to \text{Set}$$
$$\mathcal{R}_{\mathcal{G}}[\![r]\!](t_1, t_2) :\Leftrightarrow \forall (v_1, v_2 : \cdot \vdash r).t_1 \rightarrow^* v_1 \wedge t_2 \rightarrow^* v_2 \Rightarrow v_1 = v_2$$

In some sense it resembles the LR at Unit and Bool types in Figure A.6.

$$\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![\ldots]\!](t_1,t_2) :\Leftrightarrow \ldots$$
$$\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![\mathsf{Eff}_C\, A]\!](t_1,t_2) :\Leftrightarrow \forall(u_1,u_2 : \cdot \vdash \epsilon(A))(\theta_1,\theta_2,\theta_1',\theta_2' : \Sigma).$$
$$\theta_1 =_{S_{\mathtt{Atk}}} \theta_2 \wedge \theta_1, t_1 \rightsquigarrow^* \theta_1', \mathsf{return}\, u_1 \wedge \theta_2, t_2 \rightsquigarrow^* \theta_2', \mathsf{return}\, u_2$$
$$\Rightarrow \mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![A]\!](u_1,u_2) \wedge \theta_1' =_{S_{\mathtt{Atk}}} \theta_2'$$

Figure A.8.: Logical relation for λ_2^{STORE}-$\lambda_{\text{SC}}^{\text{STORE}}$ (omitting the unchanged clauses of Figure A.6)

Stores are parameterized by store typings that determine the type of the contents at each location. Since stores neither grow nor shrink, assumption (1) (Section A.3.2), we define an indistinguishability relation for stores of the same store typing. Indistinguishability is parameterized by a subset of the locations.

Definition A.4.4 (Store indistinguishability)**.** Let $S \subseteq \mathit{Loc}$. Two stores θ_1 and θ_2 of store typing $\Sigma(s)$ are S-indistinguishable, denoted by $\theta_1 =_S \theta_2$, if they are indistinguishable at each location in S, i.e. $\forall s \in S.\ \mathcal{R}_{\mathcal{G}}[\![\Sigma(s)]\!](\theta_1(s),\theta_2(s))$.

NI for λ_2^{STORE} programs is:

Definition A.4.5 (NI for λ_2^{STORE}-$\lambda_{\text{SC}}^{\text{STORE}}$)**.** A program $\Sigma, \cdot \vdash p : \mathsf{Eff}_S\, \mathsf{Unit}$ satisfies NI *with respect to* $S' \subseteq \mathit{Loc}$, if for any two stores $\theta_1, \theta_2 : \Sigma$, any two values $\cdot \vdash v_1, v_2 : \mathsf{Eff}_S\, \mathsf{Unit}$, and any two stores $\theta_1', \theta_2' : \Sigma$, if $\theta_1 =_S \theta_2$ and $\theta_1, p \rightsquigarrow^* \theta_1', v_1$ and $\theta_2, p \rightsquigarrow^* \theta_2', v_2$ then $\theta_1' =_S \theta_2'$.

Again, $\lambda_{\text{SC}}^{\text{STORE}}$ enforces NI on λ_2^{STORE} programs:

Theorem A.4.5. *Given any* λ_{SC}^{STORE} *term* $\{\mathtt{Atk}\}\ ;\ \Sigma, \cdot \vdash^{SC} t : \mathsf{Eff}_S\, \mathsf{Unit}$ *the erased program* $\Sigma, \cdot \vdash \epsilon(t) : \mathsf{Eff}_S\, \mathsf{Unit}$ *satisfies NI* with respect to $S_{\mathtt{Atk}}$ *where* $S_{\mathtt{Atk}} := \{s \mid s \in \mathit{Loc}, \mathrm{label}(s) \sqsubseteq \mathtt{Atk}\}$.

The LR that we construct to prove NI (Figure A.8) is largely similar to that of λ_2^{PRINT} (Figures A.7 and A.6), with the difference that effectful programs take as argument and produce as result indistinguishable pairs of stores. NI follows as a consequence of the fundamental theorem of the LR.

A.5. Implementation

In this section, we present an implementation of SC and $\lambda_{\text{SC}}^{\text{PRINT}}$ (Sections A.2 and A.3.1) as a Haskell library, which we call SCLib. We omit $\lambda_{\text{SC}}^{\text{STORE}}$ (Section A.3.2) for lack of space. However, its implementation is similar to that

of $\lambda_{\text{SC}}^{\text{PRINT}}$. Furthermore, we demonstrate that existing Haskell libraries for static IFC can be reimplemented in terms of the interface that SCLib exposes.

The main characteristic of SC is that typing judgements $\pi \,;\, \Gamma \vdash^{\text{SC}} t : A$ are indexed by a set of labels π. Onwards, we refer to the left part of the judgement, i.e. $\pi \,;\, \Gamma$, as the context of the term t. The set of labels plays an important role in enforcing IFC. However, individual labels are not first-class citizens: there is no type of labels and, hence, labels can neither be introduced nor eliminated. Further, some typing rules in SC modify the set of labels in their context: e.g. the rule for unseal_l (cf. Figure A.1) augments the set in its premise with l. When shallowly embedding in Haskell any calculus that manipulates the context in this fashion, there is a natural problem to overcome: Haskell does not allow library implementors to have access to a program's context. For instance, to embed a linear type system, Bernardy et al. (2017) need to change the compiler.

To overcome these difficulties, our implementation resorts to a combination of Haskell's module system to hide the implementation details' from users, and the use of Haskell's function space, i.e. abstraction and application, to manage the runtime representation of labels. We hope to convince the reader that our simple implementation matches the studied enforcement mechanisms, and that it can shed light on previous work on static IFC in Haskell.

A.5.1. Implementation of SC

When we introduced SC, we mentioned the intuition that labels in π are some sort of type-level keys whose possession permits access to sealed data. Our implementation takes this intuition literally: there is a type for keys whose elements are attached with type-level labels, and the primitive to unseal, i.e. unseal, is parameterized by a key. The elements of this type are like capabilities (Dennis and Horn 1983) which need to be explicitly exercised.

Figure A.9 shows the *complete* implementation of SC in Haskell. Without loss of generality, we assume the two-point security lattice. As previous work (e.g. MAC (Russo 2015), HLIO (Buiras et al. 2015), and DCC (Algehed and Russo 2017)) we represent labels as types of kind `Label` (line 5, and the use of the GHC extension `DataKinds`) and encode the "flows to" relation via a typeclass (lines 7–10). For simplicity we show the encoding of the two-point security lattice, however, this can be generalized (cf. (Buiras et al. 2015)). In line 12 we introduce a new datatype `Key` which is parameterized by a type `l` of kind `Label`. Then, line 14 introduces the type `S`, which is a wrapper over the function space between the types `Key l` and `a`, i.e. `Key l -> a`.

We now implement the primitives seal and unseal—Rules Seal and Unseal

```
module SCLib
  (Key (), Label (..), FlowsTo (..), S (S), seal, unseal, ...)
where
-- Enumeration of security labels for the two-point lattice
data Label = H | L
-- "Flows to" relation as a typeclass
class FlowsTo (l :: Label) (l' :: Label)
-- Instances
instance FlowsTo l l
instance FlowsTo L H
-- Type-level keys
data Key (l :: Label) = Key
-- Security type
data S l a = S (Key l -> a)
-- Sealing
seal :: (Key l -> a) -> S l a
seal = Seal
-- Unsealing
unseal :: FlowsTo l' l => Key l -> S l' a -> a
unseal k@Key (S f) = f Key
```

Figure A.9.: Implementation of SC (e.g. for the two-point security lattice)

from Figure A.1. Rule Seal introduces a sealed value of type `S l a` from a value of type `a` that is typed in a context that has been extended by label `l`. Our implementation (lines 16 and 17), however, uses the function space `Key l -> a` to mimic the context extension by label `l`. Intuitively, the value of type `Key l` represents a proof that the label `l` is present in the context, and hence it can be used to construct the value of type `a` through `unseal`, which we explain next. Rule Unseal permits to eliminate sealed values of type `S l a` provided that the context contains a label `l'` secret enough, i.e. `l FlowsTo l'`. The combinator `unseal` (lines 19–20) allows unsealing terms of type `S l a` precisely in case we *have* a value of type `Key l'` and the label in its type, i.e. `l'`, flows to `l`—see the constraint `FlowsTo l' l`. In order to enforce IFC, it is important that the constructors of `Key` are kept abstract from the user—observe `Key ()` in the export list of the module (line 2). Otherwise, anyone could extract the underlying term of type `a` from an `l`-sensitive value `secret :: S l a` by applying `unseal Key`. Similar to Russo et al. (2008), the combinator `unseal` is strict in its argument (`k@Key`) in order to forbid forged keys like `undefined :: Key l`. We remark that the noninterference property—recall termination-insensitive noninterference (TINI) from Definition A.4.1—rules out programs that force `undefined` and halt with error.

The implementation discussed so far consists of the trusted computing base (TCB) of SCLib. From now on, users of the library can derive functionalities from the library's interface. For example, programmers can implement the `Functor`, `Applicative` and `Monad` instances for the type `S l a`. For instance,

```
instance Functor (S l) where
  fmap f x = seal (\k -> f (unseal k x))

instance Applicative (S l) where
  pure x  = seal (\k -> x)
  f <*> a = seal (\k -> (unseal k f) (unseal k a))

instance Monad (S l) where
  return = pure
  m >>= f = seal (\k -> unseal k (f (unseal k m)))
```

Note that the programmer does not need access to the TCB in order to implement these instances—as opposed to MAC (cf. (Vassena, Buiras, et al. 2016)). This phenomenon, we believe, is a sign of the simplicity and generality of our implementation.

A.5.2. Implementation of λ_{SC}^{PRINT}

Figure A.10 shows the implementation of λ_{SC}^{PRINT} which builds on the implementation of SC (Figure A.9). The datatype `Eff` wraps `IO` computations and is indexed by a type-level set (D. A. Orchard and Petricek 2014) `ls` of channels where the computation can write to. For simplicity, we will omit the map label (Figure A.3) from channels to labels and identify channels with labels so that the index `ls` has kind `[Label]`. This makes the implementation simpler.

In lines 15–18 we implement the return and bind of the graded monad. `returnEff` does not produce effects, thus, its type is indexed by the empty set `[]` (cf. Figure A.3). `bindEff` type is indexed by the union of the sets of labels of the computations (cf. Figure A.3). The implementation of subeffecting is the identity function (lines 20–21). Printing effects can be performed by the combinator `printEff` (lines 26–30). The argument of type `SLabel l` is a term level representation of a the type-level label of printing channel.

Lines 34–36 show the implementation of the novel primitive `distr`. The typeclass constraint `FlowsToSet l ls` (see the definitions in lines 32) ensures that $\texttt{l} \sqsubseteq \texttt{l'}$ for every label `l'` in `ls`. Its implementation is standard. The implementation uses the value `Key`, which pertains to the TCB, to unseal the `Eff` action, i.e. `unseal Key m`; and then it runs its effects, i.e. `res <- runEff (unseal Key m)`; finally it seals the result at the same label, i.e. `return (seal (\k -> res))`.

A.5.3. Implementing Existing Libraries for IFC

We conclude this section by showing that we can reimplement some of the existing libraries in Haskell for IFC. We show implementations of SecLib (Russo et al. 2008), simplified dependency core calculus (SDCC) (Algehed 2018) (an alternative presentation of DCC) and a variation of MAC (Russo 2015)[7] using SCLib interface. In some sense the implementations help to explain the mentioned libraries. Further, this shows that the programmer can choose to write programs against SCLib's "low-level" interface; or a more "high-level" interface, e.g. MAC; or a combination of both. We declare future work to compare the performance among implementations.

For each library we briefly explain its interface and show its implementation in terms of SCLib.

SecLib and SDCC SecLib is one of the pioneers of static IFC libraries in the context of Haskell. Its main feature is a family of security monads `Sec`

[7] In our variation the type `Labeled` is a monad. This is "unsafe" in MAC (cf. (Vassena, Russo, Buiras, et al. 2018, Section 9.1)).

```
module SCLib
  (..., Eff (), FlowsToSet (), pureEff, appEff, returnEff
  , bindEff, distr, subeff, printEff)
where

newtype Eff (ls :: [Label]) a = Eff { runEff :: IO a }
-- Functor
instance Functor (Eff ls) where
  fmap f (Eff io) = Eff (fmap f io)
-- Applicative
pureEff :: a -> Eff [] a
pureEff = returnEff

appEff :: Eff ls1 (a -> b) -> Eff ls2 a -> Eff (Union ls1 ls2) b
appEff (Eff ioff) (Eff ioa) = Eff (ioff <*> ioa)
-- Graded monad
returnEff :: a -> Eff [] a
returnEff a = Eff (return a)

bindEff :: Eff ls1 a -> (a -> Eff ls2 b) -> Eff (Union ls1 ls2) b
bindEff (Eff m) f = Eff (m >>= runEff . f)
-- Subeffecting
subeff :: Subset ls1 ls2 => Eff ls1 a -> Eff ls2 a
subeff (Eff m) = Eff m
-- Print
data SLabel :: Label -> * where
  SH :: SLabel H
  SL :: SLabel L

printEff :: (Show a) => SLabel l -> a -> Eff [l] ()
printEff l x = Eff (print (header l) >> print x)
  where header :: SLabel l -> String
        header SH = "Channel H:"
        header SL = "Channel L:"
-- Distr
type family FlowsToSet (l :: Label) (ls :: [Label]) :: Constraint
...

distr :: FlowsToSet l ls => S l (Eff ls a) -> Eff ls (S l a)
distr m = Eff (do res <- (runEff (unseal Key m))
                  return (seal (\k -> res)))
```

Figure A.10.: Implementation of λ_{SC}^{PRINT} (omitting the unchanged code of Figure A.9)

indexed by labels from the security lattice, each equipped with `>>=` (bind) and `return`. SecLib's special ingredient is a combinator `up` that allows coercions from lower to higher labels in the security monad.

SDCC is an alternative presentation of DCC which favours a simple set of combinators instead of DCC's nonstandard bind and *protected at* relation. Similar to DCC, SDCC sports a family of monads indexed by security labels. These support `fmap`, `return` and `>>=` (bind). Further, SDCC implements two combinators `up` and `com`, that allow to relabel in the style of SecLib and commute terms of monadic type with different labels. SDCC's interface is strictly a superset of that of SecLib, thus we directly show the implementation of the former.

```
type T l a = S l a

instance Functor (T l) where
  ...
instance Monad (T l) where
  ...

up :: FlowsTo l l' => T l a -> T l' a
up lv = seal (\k -> unseal k lv)

com :: T l (T l' a) -> T l' (T l a)
com lv = seal (\k' -> seal (\k -> unseal k' (unseal k lv)))
```

MAC MAC which we discussed in the introduction, is one of the state-of-the-art libraries for effectful IFC in Haskell. At its core, MAC defines two types: `Labeled l a` for pure sensitive values, and `MAC l a` for secure computations. `MAC l a` is a monad for each label `l` where the label: 1. protects the data in context; and 2. restricts the permitted effects. (cf. (Vassena, Russo, Buiras, et al. 2018)) MAC's functionality stems from the interaction between `Labeled` and `MAC` through the primitives `label` and `unlabel`. In order to label a value one needs to do so within the `MAC l` monad: the `Labeled l a` type does not export a combinator `a -> Labeled l a`. Our implementation, however, permits to do so. Below we show MAC's implementation in terms of SCLib.

```
type Labeled l a = S l a

type MAC l a = forall ls. FlowsToSet l ls => Eff ls (S l a)

label :: FlowsTo l l' => a -> MAC l (Labeled l' a)
label a = returnEff (seal (\k -> a))
```

```
unlabel :: FlowsTo l l' => Labeled l a -> MAC l' a
unlabel lv = returnEff lv

join :: FlowsTo l l' => MAC l' a -> MAC l (Labeled l' a)
join m = bindEff m (\x -> seal (\k -> x))
```

A.6. Related Work

IFC for Effect-Free and Effectful Languages Algehed and Russo (2017) add effects to their embedding of DCC in Haskell but argue that their approach only works for those effects that can be implemented within Haskell. Hirsch and Cecchetti (2021) develop a formal framework based on productors and type-and-effect systems to characterize secure programs in impure languages with IFC. They give semantics to traditional security-type systems based on controlling implicit flows using program counter (PC) labels. In contrast, our approach considers from starters a pure language, where the type of effectful computation is separated from that of effect-free programs. Crary et al. (2005) present a graded monad for IFC that tracks both writes and reads on the store, while ours tracks only writes. It will be interesting to understand if this two approaches are equivalent. Devriese and Piessens (2011) add IFC mechanisms on top of existing monads for effects but don't consider the effect-free–effectful interaction.

Modalities for IFC The languages and IFC enforcement mechanisms that we present are based on the sealing calculus (SC) of Shikuma and Igarashi (2008). Differently from them, we think of SC terms as evidence that STLC programs satisfy noninterference. The work by Miyamoto and Igarashi (2004) gives an informal connection between a classical type system for IFC and a certain modal logic. Their type system is very different from our enforcement mechanism in that a typing judgement has two separate variable contexts. Recently, the work by Abel and Bernardy (2020) presents a unified treatment of modalities in typed λ-calculi. The authors present a effect-free lambda calculus parameterized by family of modalities with certain mathematical structure, and show that many programming language analyses, including IFC, are instantiations of their framework. In contrast to our work, it is not very clear how one would implement theirs system in Haskell, since it would require a fine-grained control over the variables in the context. Kavvos (2019) studies modalities for IFC in the classified sets model, which they use to prove noninterference properties for a range of calculi that includes SC.

Coeffectful Type Systems for IFC A recent line of work suggests using coeffect type systems to enforce IFC. Petricek et al. (2014) develop a calculus to capture different granularity demands on contexts, i.e. flat whole-context coeffects (like implicit parameters (Lewis et al. 2000)) or structural per-variable ones (like usage or data access patterns). The work by Gaboardi et al. (2016) expands on that and uses graded monads and comonads to combine effects and coeffects. The authors describe *distributivity laws* that are similar to our primitive distr addresses. The article suggests IFC as an application where the coeffect system captures the IFC constraints and the effect system gives semantics to effects. The distributive laws explains how both are combined. However, their work does not state neither proves a security property for their calculus. Different from it, our work does not use comonads as the underlying structure for IFC, and further considers printing and global store effects. Granule is a recent programming language (D. Orchard et al. 2019) based on *graded modal types* that impose usage constraints on the variables.

Logical Relations for Noninterference Both Heintze and Riecke (1998) and S. A. Zdancewic (2002) use logical relation arguments to prove noninterference for a simply-typed security lambda calculus. Tse and S. Zdancewic (2004) use logical relations to prove soundness of a translation from DCC (Abadi et al. 1999) to system F and obtain noninterference from parametricity. Unfortunately, their translation is unsound (cf. (Shikuma and Igarashi 2008)). Bowman and A. Ahmed (2015) fix this by using "open" logical relations show their translation from DCC to system F_ω is sound. Different from the cited work so far, Rajani and Garg (2020) use logical relations to prove noninterference for a language with references. Gregersen et al. (2021) extend the use of logical relation to prove noninterference for languages with impredicative polymorphism. Different from Rajani and Garg (2020) and Gregersen et al. (2021), we consider first-order references for simplicity. Otherwise, we would have had to utilize a step-indexed Kripke-style logical-relations model, which would have introduced technical complications. These we believe are orthogonal to the main contribution of our work.

A.7. Conclusions

In this paper, we have demonstrated that to enforce IFC in pure languages with a single primitive distr suffices to securely control what information flows from sensitive data to effects. To support our claim, we have presented IFC enforcement mechanisms for several kinds of effects and proved that they

satisfy noninterference. Our development rests on the insight that effect-free IFC for pure languages can already express that a computation will not leak sensitive data through its effects when executed. Then, a single primitive, distr, to execute these is enough to extend IFC to effects and retain the security guarantees. We hope that this work brings a new perspective to IFC research for pure languages with effects.

Acknowledgements

We would like to thank Fabian Ruch for his feedback on earlier versions of this work. We would also like to thank the anonymous referees for their valuable comments and helpful suggestions.

This work is supported by the SSF under the projects WebSec (Ref. RIT17-0011) and Octopi (Ref. RIT17-0023R).

Bibliography

Abadi, Martín et al. (1999). "A Core Calculus of Dependency". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, pp. 147–160. DOI: 10.1145/292540.292555. URL: https://doi.org/10.1145/292540.292555 (cit. on pp. 23, 55).

[SW] Abel, Andreas, Guillaume Allais, et al., *Agda 2* 2005–. Chalmers University of Technology and Gothenburg University. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php (cit. on p. 27).

Abel, Andreas and Jean-Philippe Bernardy (2020). "A unified view of modalities in type systems". In: *Proc. ACM Program. Lang.* 4.ICFP, 90:1–90:28. DOI: 10.1145/3408972. URL: https://doi.org/10.1145/3408972 (cit. on p. 54).

Ahmed, Amal J. (2006). "Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types". In: *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*. Ed. by Peter Sestoft. Vol. 3924. Lecture Notes in Computer Science. Springer, pp. 69–83. DOI: 10.1007/11693024_6. URL: https://doi.org/10.1007/11693024%5C_6 (cit. on p. 42).

Algehed, Maximilian (2018). "A Perspective on the Dependency Core Calculus". In: *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by Mário S. Alvim and Stéphanie Delaune. ACM, pp. 24–28. DOI: 10.1145/3264820.3264823. URL: https://doi.org/10.1145/3264820.3264823 (cit. on pp. 28, 51).

Algehed, Maximilian and Alejandro Russo (2017). "Encoding DCC in Haskell". In: *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2017, Dallas, TX, USA, October 30, 2017*. ACM, pp. 77–89. DOI: 10.1145/3139337.3139338. URL: https://doi.org/10.1145/3139337.3139338 (cit. on pp. 48, 54).

Bernardy, Jean-Philippe et al. (2017). "Linear Haskell: practical linearity in a higher-order polymorphic language". In: *CoRR* abs/1710.09756. arXiv: 1710.09756. URL: http://arxiv.org/abs/1710.09756 (cit. on p. 48).

Bowman, William J. and Amal Ahmed (2015). "Noninterference for free". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, pp. 101–113. DOI: 10.1145/2784731.2784733. URL: https://doi.org/10.1145/2784731.2784733 (cit. on p. 55).

Buiras, Pablo et al. (2015). "HLIO: mixing static and dynamic typing for information-flow control in Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, pp. 289–301. DOI: 10.1145/2784731.2784758. URL: https://doi.org/10.1145/2784731.2784758 (cit. on pp. 23, 40, 48).

Crary, Karl et al. (2005). "A monadic analysis of information flow security with mutable state". In: *J. Funct. Program.* 15.2, pp. 249–291. DOI: 10.1017/S0956796804005441. URL: https://doi.org/10.1017/S0956796804005441 (cit. on p. 54).

Dennis, Jack B. and Earl C. Van Horn (1983). "Programming Semantics for Multiprogrammed Computations (Reprint)". In: *Commun. ACM* 26.1, p. 29. DOI: 10.1145/357980.357993. URL: https://doi.org/10.1145/357980.357993 (cit. on p. 48).

Devriese, Dominique and Frank Piessens (2011). "Information flow enforcement in monadic libraries". In: *Proceedings of TLDI 2011: 2011 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Austin, TX, USA, January 25, 2011*. Ed. by Stephanie Weirich and Derek Dreyer. ACM, pp. 59–72. DOI: 10.1145/1929553.1929564. URL: https://doi.org/10.1145/1929553.1929564 (cit. on p. 54).

Gaboardi, Marco et al. (2016). "Combining effects and coeffects via grading". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by Jacques Garrigue et al. ACM, pp. 476–489. DOI: 10.1145/2951913.2951939. URL: https://doi.org/10.1145/2951913.2951939 (cit. on p. 55).

Goguen, Joseph A. and José Meseguer (1982). "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, pp. 11–20. DOI: 10.1109/SP.1982.10014. URL: https://doi.org/10.1109/SP.1982.10014 (cit. on p. 23).

Gregersen, Simon Oddershede et al. (2021). "Mechanized logical relations for termination-insensitive noninterference". In: *Proc. ACM Program. Lang.* 5.POPL, pp. 1–29. DOI: 10.1145/3434291. URL: https://doi.org/10.1145/3434291 (cit. on pp. 42, 55).

Hedin, Daniel and Andrei Sabelfeld (2012). "A Perspective on Information-Flow Control". In: *Software Safety and Security - Tools for Analysis and Verification*. Ed. by Tobias Nipkow et al. Vol. 33. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, pp. 319–347. DOI: 10.3233/978-1-61499-028-4-319. URL: https://doi.org/10.3233/978-1-61499-028-4-319 (cit. on p. 23).

Heintze, Nevin and Jon G. Riecke (1998). "The SLam Calculus: Programming with Secrecy and Integrity". In: *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. Ed. by David B. MacQueen and Luca Cardelli. ACM, pp. 365–377. DOI: 10.1145/268946.268976. URL: https://doi.org/10.1145/268946.268976 (cit. on p. 55).

Hirsch, Andrew K. and Ethan Cecchetti (2021). "Giving semantics to program-counter labels via secure effects". In: *Proc. ACM Program. Lang.* 5.POPL, pp. 1–29. DOI: 10.1145/3434316. URL: https://doi.org/10.1145/3434316 (cit. on p. 54).

Jones, Simon L. Peyton and Philip Wadler (1993). "Imperative Functional Programming". In: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*. Ed. by Mary S. Van Deusen and Bernard Lang. ACM Press, pp. 71–84. DOI: 10.1145/158511.158524. URL: https://doi.org/10.1145/158511.158524 (cit. on p. 31).

Katsumata, Shin-ya (2014). "Parametric effect monads and semantics of effect systems". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM,

pp. 633–646. DOI: 10.1145/2535838.2535846. URL: https://doi.org/10.1145/2535838.2535846 (cit. on pp. 27, 31, 32).

Kavvos, G. A. (2019). "Modalities, cohesion, and information flow". In: *Proc. ACM Program. Lang.* 3.POPL, 20:1–20:29. DOI: 10.1145/3290333. URL: https://doi.org/10.1145/3290333 (cit. on p. 54).

Lewis, Jeffrey R. et al. (2000). "Implicit Parameters: Dynamic Scoping with Static Types". In: *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*. Ed. by Mark N. Wegman and Thomas W. Reps. ACM, pp. 108–118. DOI: 10.1145/325694.325708. URL: https://doi.org/10.1145/325694.325708 (cit. on p. 55).

Li, Peng and Steve Zdancewic (2006). "Encoding Information Flow in Haskell". In: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. IEEE Computer Society, p. 16. DOI: 10.1109/CSFW.2006.13. URL: https://doi.org/10.1109/CSFW.2006.13 (cit. on p. 23).

Miyamoto, Kenji and Atsushi Igarashi (2004). "A modal foundation for secure information flow". In: *In Proceedings of IEEE Foundations of Computer Security (FCS)*, pp. 187–203 (cit. on p. 54).

Moggi, Eugenio (1991). "Notions of Computation and Monads". In: *Inf. Comput.* 93.1, pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4. URL: https://doi.org/10.1016/0890-5401(91)90052-4 (cit. on p. 31).

[SW] Myers, Andrew C. et al., *Jif: Java information flow* 2006. URL: https://www.cs.cornell.edu/jif (cit. on p. 23).

Orchard, Dominic et al. (2019). "Quantitative program reasoning with graded modal types". In: *Proc. ACM Program. Lang.* 3.ICFP, 110:1–110:30. DOI: 10.1145/3341714. URL: https://doi.org/10.1145/3341714 (cit. on p. 55).

Orchard, Dominic A. and Tomas Petricek (2014). "Embedding effect systems in Haskell". In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. Ed. by Wouter Swierstra. ACM, pp. 13–24. DOI: 10.1145/2633357.2633368. URL: https://doi.org/10.1145/2633357.2633368 (cit. on p. 51).

Parker, James et al. (2019). "LWeb: information flow security for multi-tier web applications". In: *Proc. ACM Program. Lang.* 3.POPL, 75:1–75:30. DOI: 10.1145/3290388. URL: https://doi.org/10.1145/3290388 (cit. on p. 23).

Petricek, Tomas et al. (2014). "Coeffects: a calculus of context-dependent computation". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3,*

2014. Ed. by Johan Jeuring and Manuel M. T. Chakravarty. ACM, pp. 123–135. DOI: 10.1145/2628136.2628160. URL: https://doi.org/10.1145/2628136.2628160 (cit. on p. 55).

Polikarpova, Nadia et al. (2020). "Liquid information flow control". In: *Proc. ACM Program. Lang.* 4.ICFP, 105:1–105:30. DOI: 10.1145/3408987. URL: https://doi.org/10.1145/3408987 (cit. on p. 23).

Rajani, Vineet and Deepak Garg (2020). "On the expressiveness and semantics of information flow types". In: *J. Comput. Secur.* 28.1, pp. 129–156. DOI: 10.3233/JCS-191382. URL: https://doi.org/10.3233/JCS-191382 (cit. on pp. 23, 40, 55).

Russo, Alejandro (2015). "Functional pearl: two can keep a secret, if one of them uses Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, pp. 280–288. DOI: 10.1145/2784731.2784756. URL: https://doi.org/10.1145/2784731.2784756 (cit. on pp. 23, 25, 40, 48, 51).

Russo, Alejandro et al. (2008). "A library for light-weight information-flow security in haskell". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, pp. 13–24. DOI: 10.1145/1411286.1411289. URL: https://doi.org/10.1145/1411286.1411289 (cit. on pp. 23, 27, 50, 51).

Sabelfeld, Andrei and Andrew C. Myers (2003). "Language-based information-flow security". In: *IEEE J. Sel. Areas Commun.* 21.1, pp. 5–19. DOI: 10.1109/JSAC.2002.806121. URL: https://doi.org/10.1109/JSAC.2002.806121 (cit. on p. 23).

Shikuma, Naokata and Atsushi Igarashi (2008). "Proving Noninterference by a Fully Complete Translation to the Simply Typed Lambda-Calculus". In: *Log. Methods Comput. Sci.* 4.3. DOI: 10.2168/LMCS-4(3:10)2008. URL: https://doi.org/10.2168/LMCS-4(3:10)2008 (cit. on pp. 27, 28, 54, 55).

[SW] Simonet, Vincent, *Flow Caml* 2003. URL: http://cristal.inria.fr/~simonet/soft/flowcaml/ (cit. on p. 23).

Stefan, Deian et al. (2011). "Flexible dynamic information flow control in Haskell". In: *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. Ed. by Koen Claessen. ACM, pp. 95–106. DOI: 10.1145/2034675.2034688. URL: https://doi.org/10.1145/2034675.2034688 (cit. on p. 23).

Tse, Stephen and Steve Zdancewic (2004). "Translating dependency into parametricity". In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*. Ed. by Chris Okasaki and Kathleen Fisher. ACM,

pp. 115–125. DOI: 10.1145/1016850.1016868. URL: https://doi.org/10.1145/1016850.1016868 (cit. on pp. 27, 29, 55).

Vassena, Marco, Pablo Buiras, et al. (2016). "Flexible Manipulation of Labeled Values for Information-Flow Control Libraries". In: *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*. Ed. by Ioannis G. Askoxylakis et al. Vol. 9878. Lecture Notes in Computer Science. Springer, pp. 538–557. DOI: 10.1007/978-3-319-45744-4_27. URL: https://doi.org/10.1007/978-3-319-45744-4%5C_27 (cit. on pp. 24, 50).

Vassena, Marco, Alejandro Russo, Pablo Buiras, et al. (2018). "MAC A verified static information-flow control library". In: *Journal of Logical and Algebraic Methods in Programming* 95, pp. 148–180. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2017.12.003. URL: https://www.sciencedirect.com/science/article/pii/S235222081730069X (cit. on pp. 51, 53).

Vassena, Marco, Alejandro Russo, Deepak Garg, et al. (2019). "From fine- to coarse-grained dynamic information flow control and back". In: *Proc. ACM Program. Lang.* 3.POPL, 76:1–76:31. DOI: 10.1145/3290389. URL: https://doi.org/10.1145/3290389 (cit. on p. 23).

Volpano, Dennis M. et al. (1996). "A Sound Type System for Secure Flow Analysis". In: *J. Comput. Secur.* 4.2/3, pp. 167–188. DOI: 10.3233/JCS-1996-42-304. URL: https://doi.org/10.3233/JCS-1996-42-304 (cit. on p. 35).

Wadler, Philip and Peter Thiemann (2003). "The marriage of effects and monads". In: *ACM Trans. Comput. Log.* 4.1, pp. 1–32. DOI: 10.1145/601775.601776. URL: https://doi.org/10.1145/601775.601776 (cit. on p. 34).

Zdancewic, Stephan Arthur (2002). *Programming languages for information security*. Cornell University (cit. on p. 55).

Appendices

A.I. The Language λ_2

$$\begin{array}{rrcl} \text{Types} & a, b & ::= & \mathsf{Unit} \mid \mathsf{Bool} \mid a \Rightarrow b \\ \text{Typing contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : a \end{array}$$

$\boxed{\Gamma \vdash t : a}$

$$\textsc{Var}\ \frac{(x : a) \in \Gamma}{\Gamma \vdash x : a} \qquad \textsc{Fun}\ \frac{\Gamma, x : a \vdash t : b}{\Gamma \vdash \lambda x.\, t : a \Rightarrow b} \qquad \textsc{App}\ \frac{\Gamma \vdash t : a \Rightarrow b \qquad \Gamma \vdash u : a}{\Gamma \vdash \mathsf{app}\, t\, u : b}$$

$$\textsc{Unit}\ \frac{}{\Gamma \vdash \mathsf{unit} : \mathsf{Unit}} \qquad \textsc{True}\ \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}} \qquad \textsc{False}\ \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}$$

$$\textsc{If}\ \frac{\Gamma \vdash t : \mathsf{Bool} \qquad \Gamma \vdash u_1 : a \qquad \Gamma \vdash u_2 : a}{\Gamma \vdash \mathsf{ifte}\, t\, u_1\, u_2 : a}$$

$\boxed{t \to u \text{ with } \cdot \vdash t : a \text{ and } \cdot \vdash u : a}$

$$\textsc{App}\ \frac{t \to t'}{\mathsf{app}\, t\, u \to \mathsf{app}\, t'\, u} \qquad \textsc{Beta}\ \frac{}{\mathsf{app}\, (\lambda x.\, t)\, u \to t[u/x]} \qquad \textsc{If}\ \frac{t \to t'}{\mathsf{ifte}\, t\, u_1\, u_2 \to \mathsf{ifte}\, t'\, u_1\, u_2}$$

$$\textsc{If-True}\ \frac{}{\mathsf{ifte}\, \mathsf{true}\, u_1\, u_2 \to u_1} \qquad \textsc{If-False}\ \frac{}{\mathsf{ifte}\, \mathsf{false}\, u_1\, u_2 \to u_2}$$

B

Information Flow and Effects via Distributive Laws

Carlos Tomé Cortiñas

Manuscript

Abstract Information-flow properties of computer programs have important applications in security—for preserving confidentiality of data, for instance. Real-world programs typically perform computational effects while processing confidential data, and therefore, it is important to have better tools to study information-flow properties of these programs. To this end, we leverage existing semantic models of information flow without effects and monads on these for modelling effects and their security specifications.

In this work, we show that computational monads on categories of classified sets yield a systematic and modular approach to reasoning about information-flow properties of programs with general effects. We show that the usual operation of "labelling", which is pervasive in type systems and models of information flow, is closely related to the secure execution of effects, that is, whether it is secure to eliminate labelled values, i.e. "secrets", into effectful computations. We show that this "interaction" property naturally arises as a distributive law of labelling over effects. In fact, we prove that it is a property of semantic labelling and monads for effects whether such a (necessarily unique) distributive law exists. This key result is based on abstract properties of semantic labelling and monads, and whence, it is broadly applicable in other well-known models of information flow.

B.1. Introduction

In this paper we are concerned with providing a solid *semantic foundation* for studying information-flow properties of higher-order computer programs *with general effects*.

Information-flow properties of computer programs have important applications in language-based security to help protect confidentiality and integrity aspects of data as it is processed by programs. In programming languages without effects, e.g. pure functional languages, these information-flow properties usually concern the extensional input–output behaviour of programs. Confidentiality policies commonly forbid flows of information from **secret** sources to **public** sinks. An information-flow property for enforcing this policy may read as follows:

> Do outputs labelled as public depend *in an essential way* on inputs labelled as secret?

Note the emphasis on "in an essential way": programs can depend on secret data as far as it does not influence (or *leak* through) its public outputs.

The information-flow property that public outputs do not depend on secret inputs is typically formalized using noninterference (Goguen and Meseguer 1982). A program satisfies noninterference if varying its secret (or High) inputs does not affect its public (or Low) outputs. We refer to L and H as security levels. Note that noninterference depends on a program together with a security specification: that is, an assignment of security levels—or *labelling*—to inputs and outputs.

Semantically, a program f between Booleans denotes a function $f : 2 \to 2$. A security specification that labels the inputs of f with secret and its output with public is written as $2^{\mathtt{H}} \to 2^{\mathtt{L}}$. Since the policy disallows flows from H to L, noninterference for the function f w.r.t. this specification formally states that for all Booleans $b_1, b_2 \in 2$, $f(b_1) = f(b_2)$, that is, the function is constant on its input. Since the output is a Boolean equality is enough. When f satisfies noninterference, i.e. it is *secure*, we write:

$$f : 2^{\mathtt{H}} \to 2^{\mathtt{L}}$$

The denotations of security specifications and programs satisfying noninterference properties for these, e.g. like the one above, is well understood: several semantic models of information flow have been proposed in the past—e.g. Sterling and Harper (2022), Abadi et al. (1999), Kavvos (2019), and Abramsky and Jagadeesan (2009). In particular, these models explain the operation of

"labelling" as a family $\blacklozenge$ of modalities indexed by security levels, e.g. $\blacklozenge_{\mathtt{H}}$ or $\blacklozenge_{\mathtt{L}}$ acting on semantic types. Common to all these models is the structure of labelling: namely that of a family of idempotent monads together with certain monad morphisms that "follow" the permitted flows. In some sense this structure enforces that flows of information respect the intended policies: e.g. from types labelled at $\mathtt{L}$ to those labelled at $\mathtt{H}$, but not the other way around. On the practical side, type systems for information flow incorporate these modalities in the form of (syntactic) labelling of types and terms—see e.g. Rajani and Garg (2020).

Programs in real-world programming languages typically do not run in isolation, just as purely input–output processes, but interact in nontrivial ways with the environment in which they run, e.g. the user or other programs running in the system or elsewhere. In addition to possibly returning a final value these programs may *have an effect* on the "world" by performing effectful operations which usually cannot be undone, e.g. printing on a channel or modifying the memory. Information-flow properties for these programs need to take into account effects since they can be (ab)used to leak information, and, in fact, these properties are concerned not only with their input–output behaviour but also with the *effectful behaviour* of programs. An information-flow property for effectful programs thus may read as:

> Do outputs labelled as public *and publicly observable effects* depend in an essential way on inputs labelled as secret?

Note that what effects are considered "publicly observable" depends on 1. the kinds of effects and the information that can be extracted from them, and 2. the amount of information leakage that is deemed reasonable for practical purposes—see, e.g., Askarov, Hunt, et al. (2008) for the case of nontermination. We therefore conclude that labelling inputs and outputs alone is not enough to write the intended security specifications, and thus study information-flow properties, for these programs.

Moggi (1989) has shown that effectful programs between Booleans denote functions, and more generally morphisms, $g : 2 \to T(2)$, where the type constructor T in the codomain of g has the structure of a (strong) computational monad: the monad T describes the possible effects that the program may perform during its execution. Inspired by Moggi and labelling, we write a security specification for the program g using an "effect labelling" operation S on semantic types:

$$2^{\mathtt{H}} \to S(2^{\mathtt{L}})$$

Intuitively, the effect labelling S should at least capture the effects of computations—like monads do—and further describe when the effects of two

computations are to be considered indistinguishable to *public observers*. Noninterference for the program g w.r.t. this specification states that for all Booleans $b_1, b_2 \in \mathbb{2}, g(b_1) \simeq g(b_2)$: that is, f produces indistinguishable effects and returns equal Booleans, denoted by the symbol $\simeq$, for any two secret inputs. Indistinguishability is usually an "equivalence" relation weaker than equality to permit some practical leakage through effects.[1]

The denotation of security specifications and programs satisfying noninterference properties concerning *general effects* has not been as extensively studied as for programs without effects. That is, for general effects it is unclear what structure ought to give meaning to effect labellings, such as S. In this work, we show that computational monads on models of information flow, concretely classified sets (Kavvos 2019), correspond semantically to effect labellings.

To study what programs are secure w.r.t. security specifications that use both value and effect labelling, like for g above, it is necessary to understand how and when semantic labelling and effect labelling *interact*. From a theoretical perspective, understanding this interaction clarifies the scope of information leakage arising from effects. In type systems for information flow, this interaction corresponds to *elimination principles* from labelled types to computations as specified by effect labellings. In DCC-style type systems (Abadi et al. 1999), for instance, this elimination takes the following form:

$$\frac{\Gamma \vdash t : \blacklozenge_l(A) \qquad \Gamma, A \vdash u : S(B) \qquad l\text{-protected}(S(B))}{\Gamma \vdash \mathsf{let}\, t\, u : S(B)}$$

In this paper, we show that nice interactions between labelling and effect labelling correspond to distributive laws of the former over the latter. We prove that it is a property of labelling at some label l and the computational monad for effect labelling whether a (necessarily unique) distributive law exists. For example, the validity of the above rule depends on the existence of a distributive law of $\blacklozenge_l$ over S; and the fact that this law is unique justifies the extension of DCC's "protected at" relation, i.e. l-protected, to effect labelling as used above.

B.1.1. Contributions

Contribution 1: *Monads as models of effects in information flow.* In this paper we present a novel abstraction to model security specifications and noninterference properties for general effects. This abstraction, "effect labellings", are computational monads but considered not on categories

[1] In some cases, e.g. nontermination, "equivalence" might not even an equivalence relation—see Hunt et al. (2023)

of programs, e.g. the category of sets and functions, but on categories of information flow. Concretely, we present extensive evidence that shows that computational monads on the category of classified sets by Kavvos (2019) are suitable for capturing effects, describing security specifications, and specifying information-flow properties of effectful programs.

This enables our second and most important contribution.

Contribution 2: *Interaction between labelling and monads for effects.* Modelling effects and their security specifications by computational monads on classified sets leads to the question of when and how labelling and these monads interact. In this paper, we show that this interaction naturally arises in the form of distributive laws. Further, we show that once effects have been specified by a computational monad S then it is a *property* of labelling at some security level l and S whether such a distributive law exists, and hence, elimination principles from labelled values at l to effects as specified by S are secure. In practice this means that once effects are specified there is a limitation on what effects can depend on labelled types, and there is not much else that can be done.

Remark B.1.1. This paper is written in the (basic) language of category theory. The motivation is twofold: to reuse the existing theory, which has been developed in this terms; and, to remain as model independent as possible from the particularities of classified sets. In fact, Contribution 2 is not specific to the model of classified sets: it depends solely on 1. using monads for modelling information flow with effects, and 2. modelling labelling via a family of idempotent monads indexed by labels. Since (2) is a common feature of other known models of information flow—i.e. Hunt et al. (2023), Sterling and Harper (2022), Abadi et al. (1999), and Abramsky and Jagadeesan (2009)—the observation in Contribution 2 is also applicable in those situations.

B.1.2. Outline

The rest of this paper is structured as follows:

- Section B.2 reviews categories of classified sets as models of information flow.
- Section B.3 recalls the operation of "labelling" on categories of classified sets, which onwards we call *redaction*, and its properties in terms of information flow.

- Section B.4 shows that computational monads on categories of classified are suitable abstractions to give semantics to "effect labellings", that is, to encode and reason about information-flow properties of programs with general effects. This section treats the interaction of computational monads for effects (on classified sets) and redaction, and proves that, in fact, when this interaction exists it is necessarily unique and corresponds to certain distributive law from redaction to effects.
- Section B.5 shows that the effect of divergence and information-flow properties regarding this effect also fit in the framework as outlined in Section B.4.
- Section B.6 reviews related work, and
- Section B.7 concludes this paper with some directions for further work.

As a pointer to the reader, Sections B.4 and B.5 contain the main contributions of the present work, namely, Contributions 1 and 2.

B.2. Preliminaries on Classified Sets

In this section we recall some basic concepts and properties of the category of classified sets over some security lattice (Kavvos 2019). This category is a more high-tech variation of the model presented by Abadi et al. (1999) in their work on dependency analysis and DCC. The reader already familiar with this might skip the section and jump directly to Section B.3.

Let $\mathcal{L} = (L, \sqsubseteq, \bot, \vee)$ be a join semilattice, where $\sqsubseteq$ is the partial order relation, pronounced "flows to", $\bot$ the least element, and $\vee$ the binary join. We think of $\mathcal{L}$ as a security lattice in the sense of Denning (1976) and refer to its elements $l \in L$ as security levels.

Definition B.2.1. A classified set A over $\mathcal{L}$ (or $\mathcal{L}$-classified set) consists of

(1) a carrier set $\mathrm{U}(A)$, and

(2) a family of reflexive binary relations $\mathrm{R}_l(A)$ on $\mathrm{U}(A)$ indexed by levels $l \in L$.

A binary relation at level l, i.e. $\mathrm{R}_l(A)$, formalizes the view that observers at l have on values of the carrier set, i.e. A. We call $\mathrm{R}_l(A)$ the "indistinguishability" relation at l, and we say that two elements $a_1, a_2 \in \mathrm{U}(A)$ are indistinguishable at l iff $\mathrm{R}_l(A)(a_1, a_2)$. Note that this Definition imposes no restrictions other

than reflexivity on the relations, in particular, e.g., $\mathrm{R}_l(A)$ need not be an equivalence relation.

To make things more concrete. Let $\mathtt{LH} \coloneqq \mathtt{L} \sqsubseteq \mathtt{H}$ be a two-point lattice where $\mathtt{L}$ stands for *public* and $\mathtt{H}$ for *secret*.

Example B.2.1. The classified sets over $\mathtt{LH}$ that correspond to the type of secret and public Booleans, $2^{\mathtt{H}}$ and $2^{\mathtt{L}}$, are

$$\begin{aligned} \mathrm{U}(2^{\mathtt{H}}) &\coloneqq 2 & \qquad \mathrm{U}(2^{\mathtt{L}}) &\coloneqq 2 \\ \mathrm{R}_{\mathtt{L}}(2^{\mathtt{H}})(x,y) &\coloneqq \top & \qquad \mathrm{R}_{\mathtt{L}}(2^{\mathtt{L}})(x,y) &\coloneqq x = y \\ \mathrm{R}_{\mathtt{H}}(2^{\mathtt{H}})(x,y) &\coloneqq x = y & \qquad \mathrm{R}_{\mathtt{H}}(2^{\mathtt{L}})(x,y) &\coloneqq x = y \end{aligned}$$

where $2 \coloneqq \{tt, tt\}$ is a two-point set. Note in the definition of $2^{\mathtt{H}}$ that public observers, i.e. at level $\mathtt{L}$, are not permitted to make distinctions between the Booleans tt and tt when regarded as secret. In contrast, all observers are allowed to distinguish among the public Booleans.

The family of relations $\mathrm{R}(A)$ of a classified set A is to be construed as part of the security specification of a system: an indistinguishability relation $\mathrm{R}_l(A)$ specifies what distinctions between values of type $\mathrm{U}(A)$ actually count as conveying information to observers at level l or above. For example, a program $p : \mathrm{U}(A)$ that in two different executions returns two l-related values $a_1, a_2 \in \mathrm{U}(A)$ conveys the same amount of information as a program $p' : \mathrm{U}(A)$ that always returns, say, a_1. This relation should not be confused with what data "real world" observers can actually observe (and thus distinguish) by observing programs in the system. Instead it corresponds to the kind of *information leaks* that are deemed reasonable—for instance for practical purposes—according to certain threat model.

For instance,

Example B.2.2. Let A be a classified set over $\mathtt{LH}$. Define a classified set $\mathrm{Partial}^I(A)$ by

$$\begin{aligned} \mathrm{U}(\mathrm{Partial}^I(A)) &\coloneqq \mathrm{U}(A) + 1 \\ \mathrm{R}_l(\mathrm{Partial}^I(A))(x,y) &\coloneqq \begin{cases} \mathrm{R}_l(A)(x',y') & x = \mathrm{inl}(x') \wedge y = \mathrm{inl}(y') \\ \top & \text{otherwise} \end{cases} \end{aligned}$$

If a value of type $\mathrm{Partial}^I(A)$ is thought of as the result of a computation that either returns a value of type A or fails, then the family of relations specifies that computations are permitted to leak (one bit of) information

through the "termination channel" but none via their final value. For example when A is $2^{\mathtt{L}}$, $\mathrm{R}_{\mathtt{L}}(\mathrm{Partial}^I(2^{\mathtt{L}}))(\mathrm{inl}(tt), \mathrm{inr}(\star))$ but it is not the case that $\mathrm{R}_{\mathtt{L}}(\mathrm{Partial}^I(2^{\mathtt{L}}))(\mathrm{inl}(tt), \mathrm{inl}(tt))$

Note that the relation $\mathrm{R}_l(\mathrm{Partial}^I(A))$ is reflexive and symmetric but not necessarily transitive. If it is transitive, then it is the everywhere true relation (i.e. relating any two pairs of elements). In fact the relation $\mathrm{R}_l(\mathrm{Partial}^I(A))$ is transitive if and only if the relation $\mathrm{R}_l(A)$ is everywhere true.

The type $\mathrm{Partial}^I(A)$ encodes an information-flow property known as (batch) termination *I*nsensitive noninterference. Typically type systems for information flow enforce this property when partiality arises from nontermination in the underlying programming language— see, e.g., Askarov, Hunt, et al. (2008).

Definition B.2.2. A map f between classified sets A and B (over $\mathcal{L}$) consists of a function $\mathrm{U}(f) : \mathrm{U}(A) \to \mathrm{U}(B)$ that preserves indistinguishability at every level, i.e. for all $l \in L$, $\mathrm{R}_l(B)(f(a_1), f(a_2))$ whenever $\mathrm{R}_l(A)(a_1, a_2)$.

Example B.2.3. The *function* space $2 \to 2$ consists of the identity, negation, and constant functions $\lambda x.tt$ and $\lambda x.tt$. Of these four functions only the last two are maps of $\mathtt{LH}$-classified sets $2^{\mathtt{H}} \to 2^{\mathtt{L}}$, i.e. the maps between $2^{\mathtt{H}}$ and $2^{\mathtt{L}}$ are exactly the noninterferent functions; the other two functions, i.e. identity and negation, are *interferent*. On the other hand, note that all functions in $2 \to 2$ are maps of $\mathtt{LH}$-classified sets $2^{\mathtt{L}} \to 2^{\mathtt{H}}$. In particular, this means that $2^{\mathtt{L}} \ncong 2^{\mathtt{H}}$.

For any security lattice $\mathcal{L}$, classified sets over $\mathcal{L}$ and maps between them form a category $\mathrm{CSet}_{\mathcal{L}}$. This category has a rich type structure:

Theorem B.2.1 (cf. Kavvos (2019, Theorem 2))**.** $\mathrm{CSet}_{\mathcal{L}}$ *is bicartesian closed: it has all finite products, finite coproducts, and exponentials.*

U extends to a "forgetful" functor $\mathrm{U} : \mathrm{CSet}_{\mathcal{L}} \to \mathrm{Set}$ from classified sets (over $\mathcal{L}$) to Set which enjoys some nice properties:

Theorem B.2.2 (cf. Kavvos (2019, Theorem 6 and Corollary 3))**.** U *preserves finite limits and finite colimits.*

However,

Remark B.2.1. U *does not* preserve exponentials.

Proof. Recall that in $\mathrm{CSet}_{\mathcal{L}}$, the underlying set of the exponential object of $\mathcal{L}$-classified sets A and B is defined by $\mathrm{U}(A \Rightarrow B) \coloneqq \mathrm{Hom}_{\mathrm{CSet}_{\mathcal{L}}}(A, B)$—see Kavvos (2019, Section 2.2). In the two-point lattice $\mathtt{LH}$, the set $\mathrm{U}(2^{\mathtt{H}} \Rightarrow 2^{\mathtt{L}})$ contains two functions, $\lambda x.tt$ and $\lambda x.tt$ (cf. Example B.2.3), while the set $\mathrm{U}(2^{\mathtt{H}}) \Rightarrow \mathrm{U}(2^{\mathtt{L}})$ is the same as the set $2 \Rightarrow 2$ (by definition). □

Classified sets as refinements of sets Intuitively we can think of a classified set A as a *type refinement*—in the sense of Melliès and Zeilberger (2015)—of its carrier set $\mathrm{U}(A)$ by a family $\mathrm{R}(A)$ of relations indexed by levels.

Example B.2.4. $2^{\mathtt{H}}$ and $2^{\mathtt{L}}$ are type refinements of the set 2.

The morphisms in $\mathrm{Hom}_{\mathrm{CSet}_{\mathcal{L}}}(A, B)$ are exactly those functions $\mathrm{U}(A) \to \mathrm{U}(B)$ that satisfy the property of preserving indistinguishability relations at all $l \in \mathcal{L}$.

In fact,

Theorem B.2.3 (cf. Kavvos (2019, Theorem 3 and Corollary 1))**.** U *is faithful, i.e. the action of* U *on morphisms is injective.*

Note that, however,

Remark B.2.2. U *is not* full. Intuitively, not every function in Set preserves indistinguishability—see Example B.2.3.

For any set X, classified set B, and function $f : X \to \mathrm{U}(B)$ we can "calculate" the coarsest family of indistinguishability relations R_l (for $l \in L$) on X for which there is a (necessarily unique) $f' \in \mathrm{Hom}_{\mathrm{CSet}_{\mathcal{L}}}((X, \mathrm{R}), B)$. Analogously, for a function $f : \mathrm{U}(B) \to X$ we can calculate the finest family of relations such that $f' \in \mathrm{Hom}_{\mathrm{CSet}_{\mathcal{L}}}(B, (X, \mathrm{R}))$. This can be succinctly summarized in the terminology of *fibrations* (see, e.g., Jacobs (2001)) as:

Theorem B.2.4. $\mathrm{U} : \mathrm{CSet}_{\mathcal{L}} \to \mathrm{Set}$ *is a bifibration.*

B.3. Modalities and Information Flow: Redaction

Type systems for information flow typically come equipped with a family of modalities for labelling pieces of data with security levels. The typing rules enforce that flows of information occurring in programs are secure. For a given security level $l \in L$ this modality consist of an operation on types $\blacklozenge_l : \mathrm{Type} \to \mathrm{Type}$, pronounced "redaction at l", with typing rules for introducing and eliminating redacted values,

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{return}\, t : \blacklozenge_l(A)}\ \blacklozenge_l\text{-}\textsc{Intro}$$

$$\frac{\Gamma \vdash t : \blacklozenge_l(A) \qquad \Gamma, A \vdash u : B \qquad \blacklozenge_l\text{-protected}(B)}{\Gamma \vdash \mathsf{let}\, t\, u : B}\ \blacklozenge_l\text{-}\textsc{Elim}$$

and some computational rules (which we omit for simplicity).

The introduction principle $\blacklozenge_l$-Intro states that any term of type A can be labelled with any security level $l \in L$. The elimination principle $\blacklozenge_l$-Elim says that redacted terms of type A at level l can be eliminated into any other type B as long as B is "good enough". The abstract predicate $\blacklozenge_l$-protected(B) in the premise of this rule captures the concept of a type being good enough.

The following are examples of this (syntactic) phenomena—which is not, in any way, restricted to information flow:

Example B.3.1.

1. In the seminal work by Abadi et al. (1999) on DCC, redaction at l is given by the monad T_l and the $\blacklozenge_l$-protected predicate by the "protected at l" relation (see Abadi et al. (1999, Section 3.1) or Kavvos (2019, Section 6.1)).

2. In Moggi's Monadic Metalanguage (Moggi 1991) redaction at l is simply given by the abstract monad T (forgetting the level l) and the predicate $\blacklozenge_l$-protected(B) by "B is of the form $T(B')$" for some type B'—this example is also discussed in Kavvos (2019, Section 4.2).

3. A slight generalization of the previous example is Levy's Call-by-Push-Value (CbPV) (Levy 2004) where redaction at l is interpreted by the (abstract) monad T for effects and the predicate $\blacklozenge_l$-protected(B) by "B is a computation type".

We define (semantic) redaction at l as a monad on the category of classified sets over $\mathcal{L}$:

Definition B.3.1. Redaction at $l \in L$ is a monad $\blacklozenge_l \coloneqq (\blacklozenge_l, \eta^l, (-)^l)$ on $\mathrm{CSet}_{\mathcal{L}}$ as defined by:

$$\begin{aligned}
\mathrm{U}(\blacklozenge_l(A)) &\coloneqq \mathrm{U}(A) \\
\mathrm{R}_{l'}(\blacklozenge_l(A))(a_1, a_2) &\coloneqq \begin{cases} \mathrm{R}_{l'}(A)(a_1, a_2) & l \sqsubseteq l' \\ \top & l \not\sqsubseteq l' \end{cases} \\
\mathrm{U}(\eta^l_A) &\coloneqq \mathrm{id}_{\mathrm{U}(A)} \\
\mathrm{U}(f^l) &\coloneqq \mathrm{U}(f) \quad (\text{for } f : A \to \blacklozenge_l(B))
\end{aligned}$$

Proposition B.3.1. *For all $l \in L$, $\blacklozenge_l$ is a monad.*

Note that applying the redaction monad at l to a classified set A forces all elements of the carrier set $\mathrm{U}(A)$ to be pairwise indistinguishable at all levels

that are *not* above l. These include but is not restricted to those levels that are strictly below l since the ordering $\sqsubseteq$ is not required to be total. Intuitively, the monad *redacts* the information that could be conveyed to observers that can distinguish unrelated values of type A at security level l .

Example B.3.2. In the two-point lattice $\mathtt{LH}$ the classified set $\blacklozenge_{\mathtt{H}}(2^{\mathtt{L}})$ is the *same* as $2^{\mathtt{H}}$ (by definition).

The redaction monad at l, for any $l \in L$, enjoys some nice properties:

Proposition B.3.2.

1. $\blacklozenge_l$ *is strong.*
2. $\blacklozenge_l$ *preserves limits.*
3. $\blacklozenge_l$ *does not, in general, preserve colimits.*
4. $\blacklozenge_l$ *is idempotent: i.e. for all* $f : \blacklozenge_l(A) \to \blacklozenge_l(B)$, $(f \circ \eta^l_A)^l = f$.

Proof.

1. See Kavvos (2019, Section 5.2).
2. The functor $\blacklozenge_l$ has a left adjoint $\square_l$, pronounced "reveal at l" that for a classified set A forces the relation $\mathrm{R}_{l'}(\square_l(A))$ to be the identity relation (i.e. equality) at all levels $l' \not\sqsubseteq l$.
3. See Remark B.3.2 below.
4. Immediate by definition of $\blacklozenge_l$, and functoriality and faithfulness of U (Theorem B.2.3).

□

It is well-known that an idempotent monad $T \coloneqq (T, \eta, (-)^T)$ on a category $\mathcal{C}$ corresponds to a reflective (full) subcategory consisting of those objects $A \in \mathrm{Obj}(\mathcal{C})$ such that

$$\eta_A : A \xrightarrow{\sim} T(A) \tag{B.1}$$

is an isomorphism, i.e. η_A has a left inverse. (Borceux 1994, Vol II, Section 4.2.4)

Remark B.3.1. Note that for any $A \in \mathrm{Obj}(\mathcal{C})$, the object $T(A)$ is in the subcategory that corresponds to T, i.e. $\eta_{T(A)} : T(A) \xrightarrow{\sim} T(T(A))$.

Proof. The inverse of $\eta_{T(A)} : T(A) \to T(T(A))$ is given by

$$(\mathrm{id}_{T(A)})^T : T(T(A)) \to T(A)$$

□

In the context of classified sets,

Lemma B.3.1. *A classified set A satisfies* (B.1) *w.r.t. $\blacklozenge_l$ if and only if for all $l \not\sqsubseteq l'$ the indistinguishability relation at l' is the everywhere true relation, i.e. it relates any pair of elements.*

This motivates the following definition:

Definition B.3.2. A classified set A is l-protected iff $\mathrm{R}_{l'}(A)(a_1, a_2)$ for all $a_1, a_2 \in \mathrm{U}(A)$ and $l' \in L$ such that $l \not\sqsubseteq l'$.

Evidently any map $f : A \to B$ between classified sets A and B is a map $f : A \to B$ when A and B are regarded as l-protected classified sets. Hence, let us denote by $\mathrm{PSet}^l_{\mathcal{L}}$ the full subcategory of l-protected classified sets.

In fact—and this is typical of idempotent monads:

Proposition B.3.3. $\mathrm{PSet}^l_{\mathcal{L}}$ *is the category of algebras of $\blacklozenge_l$.*

The property of being l-protected is closed under some type operations in $\mathrm{CSet}_{\mathcal{L}}$:

Proposition B.3.4. *For all classified sets $A, B \in \mathrm{Obj}(\mathrm{CSet}_{\mathcal{L}})$,*

1. *The type 1 is l-protected.*
2. *If A and B are l-protected so is $A \times B$.*
3. *If B is l-protected so is $A \Rightarrow B$.*

These closure properties correspond to type structure in $\mathrm{PSet}^l_{\mathcal{L}}$:

Proposition B.3.5. $\mathrm{PSet}^l_{\mathcal{L}}$ *is Cartesian closed.*

However, note that the property of being l-protected is not closed under coproducts in $\mathrm{CSet}_{\mathcal{L}}$:

Remark B.3.2. It is not the case that for all $l \in L$ and $A, B \in \mathrm{Obj}(\mathrm{PSet}^l_{\mathcal{L}})$, $A+B$ is l-protected. In the two-point lattice $\mathtt{LH}$ the unit type 1 is $\mathtt{H}$-protected—see Proposition B.3.4.1. Then $2^{\mathtt{L}} \doteq 1 + 1$ and $\blacklozenge_{\mathtt{H}}(2^{\mathtt{L}}) \doteq 2^{\mathtt{H}}$. However $2^{\mathtt{L}} \ncong 2^{\mathtt{H}}$—see Example B.2.3.

Graded Redaction The family $\blacklozenge$ of redaction monads forms a graded monad (Katsumata 2014, Section 2.2):

Proposition B.3.6.

1. *For any security levels l and l' such that $l \sqsubseteq l'$, there is a monad morphism $\mathrm{up}^{l,l'} : \blacklozenge_l \dot{\to} \blacklozenge_{l'}$ given by $\mathrm{U}(\mathrm{up}^{l,l'}_A) \coloneqq \mathrm{id}_{\mathrm{U}(A)}$.*
2. *For any security levels l and l', there is a natural isomorphism $\mu^{l,l'} : \blacklozenge_l \circ \blacklozenge_{l'} \xrightarrow{\sim} \blacklozenge_{l \vee l'}$ given by $\mathrm{U}(\mu^{l,l'}_A) \coloneqq \mathrm{id}_{\mathrm{U}(A)}$.*
3. *There is a natural isomorphism $\mathrm{Id} \xrightarrow{\sim} \blacklozenge_\bot$.*

and all the necessary diagrams commute.

Part of this corresponds to the following properties of protected sets,

Corollary B.3.1.

1. *For any $l \sqsubseteq l'$, if a classified set A is l'-protected, then A is also l-protected.*
2. *All classified sets are $\bot$-protected.*

Remark B.3.3. Remark B.3.1, Proposition B.3.5, and Corollary B.3.1.1 provide a semantic justification for the "protected at" relation in DCC (Abadi et al. 1999). (Kavvos 2019, see remark below Corollary 3)

B.4. Redaction Meets Computational Effects

Moggi, in a seminal work (Moggi 1991), proposed the use of computational monads (strong monads) as a unifying framework for giving semantics to computational effects. These effects include, to name a few, throwing errors, divergence, nondeterministic operations, or printing. Interestingly, computational effects are generic in the situation at hand, and thus, in particular it is sensible to consider them in classified sets.

In the context of classified sets over some lattice $\mathcal{L}$, a computational monad S on $\mathrm{CSet}_\mathcal{L}$ can be thought of as both describing the computational effects of programs and specifying what information leaks through effectful behaviours are permitted to observers depending on their security level. This observation is key for understanding information flow in the presence of computational effects, but it is not the whole story.

Recall from Section B.3 the family of redaction monads $\blacklozenge_l$ on $\mathrm{CSet}_\mathcal{L}$, which correspond semantically to labelling in type systems for information flow—see

Remark B.3.3. Although for any $l \in L$ the monad $\blacklozenge_l$ "lives" in the same semantic universe as the monad S, they are, in general, independent. In practice, however, it is often desirable that they interact in some way since a feature of type systems for information flow is that programs which perform "secure effects", i.e. those effect whose information leakage is deemed reasonable, can depend on redacted information.

A way to understand the interaction of redaction at some level l, i.e. $\blacklozenge_l$, and the monad S is by looking at the (semantic) typing rule $\blacklozenge_l$-Elim for eliminating redacted values instantiated to computations—i.e. maps with codomain $S(B)$ for $S \coloneqq (S, \eta, (-)^S)$:

$$\frac{\Gamma \vdash t : \blacklozenge_l(A) \qquad \Gamma, A \vdash u : S(B) \qquad \blacklozenge_l\text{-protected}(S(B))}{\Gamma \vdash \mathsf{let}\, t\, u : S(B)}\ \blacklozenge_l\text{-S-Elim}$$

This typing rule is valid whenever it is *secure* to eliminate redacted values of type A at security level l into computations of type S that return values of type B: the publicly observable behaviour of the computational effects, as specified by S, and the return value of type B do not leak more information about A that what it is permitted at l. Crucially, the validity of this typing rule relies on how the $\blacklozenge_l$-protected predicate extends to computations of type S, i.e. values of type $S(A)$.

The novel observation in this paper is that once computational effects are specified in classified sets, via a monad S, then the redaction monad at l, i.e. $\blacklozenge_l$, and S either interact or do not, and, if they do, then there is a canonical way to extend the predicate $\blacklozenge_l$-protected to computations so that the typing rule $\blacklozenge_l$-S-Elim is validated. In fact, when $\blacklozenge_l$ and S interact, we extend the definition of the predicate $\blacklozenge_l$-protected with the clause:

$$\frac{\blacklozenge_l\text{-protected}(A)}{\blacklozenge_l\text{-protected}(S(A))}$$

The semantic justification for this definition, and the fact that it is canonical, comes from the following. Recall that in classified sets the predicate $\blacklozenge_l$-protected is interpreted by the semantic notion of a classified set A being l-protected—see Definition B.3.2 and Remark B.3.3—and that this further corresponds to A being an algebra for the monad $\blacklozenge_l$, see Proposition B.3.3. Then,

Theorem B.4.1. *Let $S \coloneqq (S, \eta, (-)^S)$ be a monad on $\mathrm{CSet}_{\mathcal{L}}$ and l a security level in L.*

The functor underlying S lifts to the subcategory of l-protected sets in at most one way.

Recall that a lifting of S to $\mathrm{PSet}^l_{\mathcal{L}}$ is a functor $S' : \mathrm{PSet}^l_{\mathcal{L}} \to \mathrm{PSet}^l_{\mathcal{L}}$ such that the following diagram commutes,

$$\begin{array}{ccc} \mathrm{PSet}^l_{\mathcal{L}} & \xrightarrow{S'} & \mathrm{PSet}^l_{\mathcal{L}} \\ \downarrow{\scriptstyle i} & & \downarrow{\scriptstyle i} \\ \mathrm{CSet}_{\mathcal{L}} & \xrightarrow{S} & \mathrm{CSet}_{\mathcal{L}} \end{array}$$

where $i : \mathrm{PSet}^l_{\mathcal{L}} \hookrightarrow \mathrm{CSet}_{\mathcal{L}}$ is the inclusion functor from l-protected $\mathcal{L}$-classified sets to $\mathcal{L}$-classified sets.

Proof. See Marmolejo, Rosebrugh, et al. (2002, Section 3.2) with Proposition B.3.2.4 in mind. □

Intuitively this Theorem says that it is a property of the monad S whether we can soundly extend the predicate $\blacklozenge_l$-protected to computations: either the property holds or it does not hold. Further, it provides necessary and sufficient conditions to verify whether the monad S interacts with redaction at l in a rather straightforward way: that is, exactly when the object part of S restricts to the subcategory of l-protected sets.

This motivates the following definition:

Definition B.4.1. Let $l \in \mathcal{L}$ and $S \coloneqq (S, \eta, (-)^S)$ be a monad on $\mathrm{CSet}_{\mathcal{L}}$. The monad $\blacklozenge_l$ *interacts* with the monad S iff the underlying functor of S lifts to the subcategory of l-protected sets.

It is still necessary, however, to show that for a computational monad S the typing rule $\blacklozenge_l$-S-Elim is not arbitrary and that, in some sense, it is the best we can do.

B.4.1. Interaction as a Distributive Law

Interaction between redaction at $l \in L$ and a computational monad S in the sense of Definition B.4.1 manifests as the existence of a (necessarily unique) distributive law of over S. Recall that a distributive law of $\blacklozenge_l$ over S consists of a natural transformation

$$\lambda : \blacklozenge_l S \dot{\to} S\blacklozenge_l$$

satisfying certain laws w.r.t. return and Kleisli extension of both monads (see e.g. Marmolejo and Wood (2010, Theorem 8.3)).

From the perspective of information flow, this natural transformation can be understood by replacing the typing rule for eliminating redacted values at l into computations of type S, i.e. $\blacklozenge_l$-S-Elim, by the following rule:

$$\frac{\Gamma \vdash t : \blacklozenge_l(S(A))}{\Gamma \vdash \mathsf{distr}\, t : S(\blacklozenge_l(A))}\ \blacklozenge_l\text{-S-Distr-Elim}$$

Intuitively this typing rule expresses that computations of type $S(A)$ that may depend on redacted data at l can be safely "executed" by performing the effects and finally labelling at l the result of type A. Analogously to $\blacklozenge_l$-S-Elim, the validity of this rule depends on whether information at l is permitted to be leaked through the effects that the computation S may perform.

Now recall that,

Theorem B.4.2 (cf. Beck (1969, Proposition at p. 122))**.** *Let M and N be monads on a category $\mathcal{C}$.*

Distributive laws of M over N are in one-to-one correspondence with liftings of N to the category of algebras of M.

Note that it is the monad N that lifts to algebras of M not the other way around.

Therefore,

Corollary B.4.1. *There is at most one distributive law of $\blacklozenge_l$ over S.*

Proof. Combine Theorems B.4.2 and B.4.1. □

In light of this, we conclude with an alternative formulation of the interaction of $\blacklozenge_l$ and a specification monad S.

Proposition B.4.1. *The monads $\blacklozenge_l$ and S interact iff there is a distributive law of $\blacklozenge_l$ over S.*

Overall, this connection with distributive laws suggests that our formulation of interaction between $\blacklozenge_l$ and specification monads is not *ad hoc*.

Remark B.4.1. Recall that a distributive law from a monad M to a monad N corresponds to structure that makes the composite functor, i.e. $N \circ M$, into a monad—see Beck (1969, p. 120). In terms of information flow this distributive law can be understood as the existence of a composite monad that layers a specification monad for effects on top of redaction at some $l \in L$.

Before proceeding, we note that in the case of classified sets as evinced by the forgetful functor,

$$\begin{array}{c} \mathrm{CSet}_{\mathcal{L}} \\ {\scriptstyle \mathrm{U}}\big\downarrow \\ \mathrm{Set} \end{array}$$

(see Section B.2), there is a clear distinction between:

- computational monads on Set, which *describe* the possible effectful behaviours of programs, and
- certain monads on $\mathrm{CSet}_{\mathcal{L}}$, which further *specify* what effectful behaviours observers are allowed to distinguish depending on their security level.

Therefore it is natural to consider effects as given by computational monads on the category Set in tandem with their security specifications in the form of certain monads on $\mathrm{CSet}_{\mathcal{L}}$, which we dub *specification monads*.

The rest of this section is organised in the following way:

- in Section B.4.2 we revisit the theory of computational monads on Set, and recall monads that are typically used to give semantics to effects;
- in Section B.4.3 we show that specification monads correctly capture effects and security specifications of these effects based on the effectful behaviours that observers are permitted to distinguish. To do so, for each computational monad for effects we construct several specification monads which encode different security properties.

  For each of these specification monads we check whether the condition in Definition B.4.1 is satisfied. In case it is, by Theorem B.4.1, the typing rule $\blacklozenge_l$-S-Elim is valid for this monad and hence can be used to write programs whose *secure* effects depend on redacted information.

B.4.2. Computational Effects and Monads

Effectful behaviours are an important part of everyday programs. These behaviours arise in multiple forms: from algorithms modifying some global state to computations interacting with the external world. To capture these different notions of behaviours Moggi (Moggi 1991) put forward the framework of computational monads. A computational monad T on a category $\mathcal{C}$ consists of a mapping of objects $T : \mathrm{Obj}(\mathcal{C}) \to \mathrm{Obj}(\mathcal{C})$ so that maps of shape $f : A \to T(B)$ in $\mathcal{C}$ (i.e. with codomain $T(B)$ for some $B \in \mathrm{Obj}(\mathcal{C})$) are the what we may call "computations": in addition to possibly returning values of type A they might produce effects as described by T. The monad T comes equipped

with an operation "return" $\eta_A : A \to T(A)$ that corresponds to the trivial computation that produces no effects; and a function $(-)^T : \mathrm{Hom}_{\mathcal{C}}(A, T(B)) \to \mathrm{Hom}_{\mathcal{C}}(T(A), T(B))$, called Kleisli extensions, to sequence computations by composition. Lastly, computational monads have an operation strength $\mathrm{str}_{A,B} : A \times T(B) \to T(A \times B)$ to thread values from the context to the effects. These operations should satisfy some laws: unitality of η w.r.t. sequencing, associativity of sequencing, and naturality of strength.

The interface for monads $T \coloneqq (T, \eta, (-)^T)$ does not say how to actually perform the concrete effects that the monad describes. For this, each computational monad comes equipped with *operations* for effects. (Plotkin and Power 2003)

We now recall some examples of computational monads for effects in Set and their corresponding operations.

Exceptions A computation raising exceptions from some set E corresponds semantically to a computational monad $\mathrm{Exc} \coloneqq (\mathrm{Exc}, \eta, (-)^{\mathrm{Exc}})$. The object map is $\mathrm{Exc}(A) \coloneqq A + E$; return is the left injection $\mathrm{inl} : A \to \mathrm{Exc}(A)$; and, sequencing takes a computation $f : A \to \mathrm{Exc}(B)$ to a computation $f^{\mathrm{Exc}} : \mathrm{Exc}(A) \to \mathrm{Exc}(B)$ such that $f^{\mathrm{Exc}}(\mathrm{inl}(v)) = f(v)$ for $v : \Gamma \to \mathrm{Exc}(A)$ and $f^{\mathrm{Exc}}(\mathrm{inr}(e)) = \mathrm{inr}(e)$ for $e : \Gamma \to E$. The operation $\mathrm{raise} : E \to \mathrm{Exc}(0)$ for throwing exceptions is the right injection.

Global State A computation that while running can modify an underlying state of type S is modelled by a monad $\mathrm{St}_S \coloneqq (\mathrm{St}_S, \eta, (-)^{\mathrm{St}})$ where $\mathrm{St}_S(A) \coloneqq S \Rightarrow A \times S$. The operation return $\eta : A \to \mathrm{St}_S(A)$ couples the input value of type A with the input state $\eta \coloneqq \lambda v.\lambda s.((, a), s)$; and, sequencing computations consists of threading the state $f^{\mathrm{St}} \coloneqq \lambda m.s.f(\mathrm{fst}(m(s)))(\mathrm{snd}(m(s)))$ for $f : A \to \mathrm{St}_S(B)$. Operations for the state monad are $\mathrm{put} : S \to \mathrm{St}_S(1)$ with $\mathrm{put} \coloneqq \lambda s_1, s_2.(\star, s_1)$ and $\mathrm{get} : 1 \to \mathrm{St}_S(S)$ with $\mathrm{get} \coloneqq \lambda u, s.(s, s)$. Note that global state monads model computations extensionally: a closed computation in the state monad is equivalent to first reading from the state, and then returning a value together with a new state.

Nondeterminism Computations that nondeterministically return among several values corresponds to a monad $\mathrm{NDet} \coloneqq (\mathrm{NDet}, \eta, (-)^{\mathrm{NDet}})$. The object map NDet maps a set A to the set of nonempty finite subsets of A, i.e. $\mathrm{NDet}(A) := \mathrm{P}^+(A)$; returning embeds a value v of type A to the singleton set $\{v\}$; and, sequencing applies the function to every element of the set and collects its results $f^{\mathrm{NDet}} \coloneqq \lambda m.\bigcup_{v \in m} f(v)$. The operation for producing effects is $\mathrm{choose} : 1 \to \mathrm{NDet}(2)$ defined by $\mathrm{choose} \coloneqq \lambda u.\{tt, tt\}$.

Interactive Input–Output A computation that interactively asks the user for input of type I and produces output of type O is modelled by a monad $\mathrm{IO}_{I,O} \coloneqq (\mathrm{IO}_{I,O}, \eta, (-)^{\mathrm{IO}})$ where IO maps a set A to the set inductively defined by the following clauses:

$$\frac{v : A}{\mathrm{return}(v) : \mathrm{IO}_{I,O}(A)} \qquad \frac{o : O \qquad k : \mathrm{IO}_{I,O}(A)}{\mathrm{out}(o,k) : \mathrm{IO}_{I,O}(A)} \qquad \frac{k : I \Rightarrow \mathrm{IO}_{I,O}(A)}{\mathrm{in}(k) : \mathrm{IO}_{I,O}(A)}$$

As for all inductive types, a value of $\mathrm{IO}_{I,O}(A)$ is to be construed as a "tree" with two kinds of nodes, which correspond to the constructors in and out, and a leaf return.

Returning constructs a leaf return(a) of the tree; and sequencing takes a computation $f : A \to \mathrm{IO}_{I,O}(B)$ to a computation that traverses the input tree and applies f to each of its leaves grafting the tree. The operations for input and output effects are input $: 1 \to \mathrm{IO}_{I,O}(I)$ and output $: O \to \mathrm{IO}_{I,O}(1)$, respectively.

B.4.3. Security and Specification Monads

Computations producing effects are modelled by computational monads on the category Set. As described in Section B.4.2, computational monads describe the possible behaviours of programs; how to lift pure values to effect-free computations; how to sequence the effects of several computations; and what operations actually perform the effects. However these monads on Set are unaware of any security constraints of the system, and in particular to what kinds of distinctions observers are allowed to make depending on their security level and what information leaks are permissible.

To this end we introduce *specification monads*: these are strong monads on $\mathrm{CSet}_{\mathcal{L}}$ such that, intuitively, when the security information is forgotten by U one recovers an approximation of the computational monad describing the effects—see Remark B.4.3 below. Different specification monads model different information-flow properties for programs with effectful behaviours. That there are several "reasonable" specification monads that approximate the same computational effect corresponds to the tension between properties that capture perfect, but impractical, security, and imperfect, but practical, security. On the practical side this is related to how restrictive type systems that enforce particular information-flow properties are, i.e. how many secure programs can be expressed.

This discussion motivates the following definition:

Definition B.4.2. A *specification monad* is a strong monad $S \coloneqq (S, \eta, (-)^S)$ on $\mathrm{CSet}_{\mathcal{L}}$.

In the rest of this section we construct several specification monads for each computational effect that was mentioned in Section B.4.2, and discuss how these monads model different information-flow properties of programs with effects. Most of the specification monads defined in this section correspond to security properties that have already been studied in some form or another in the language-based security literature. For each specification monad S we check for what security levels in L the monad S satisfies the condition in Definition B.4.1. For those levels $l \in L$ that S indeed satisfies the condition, we directly know that the typing rule $\blacklozenge_l$-S-Elim is valid, and hence, it is secure to eliminate redacted values at l into effects: the effects produced by the computation do not leak more information than what it is allowed at level l.

Before we proceed, some remarks.

Remark B.4.2. Recall that each computational effect comes with its own set of operations for producing effects—see Section B.4.2. Thus, specification monads for a particular effect should also include these operations.

Remark B.4.3. Intuitively, one may think of specification monads w.r.t. computational monads as liftings of monads from Set to $\mathrm{CSet}_{\mathcal{L}}$ through U—see, e.g., Kammar and McDermott (2018, Section 4.3). However, this analogy is incomplete since U does not preserve exponentials (cf. Remark B.2.1).

Remark B.4.4. Since U is faithful (cf. Theorem B.2.4), to construct a specification monad S starting with a computational monad T on Set, it is enough to provide an object mapping $S : \mathrm{Obj}(\mathrm{CSet}_{\mathcal{L}}) \to \mathrm{Obj}(\mathrm{CSet}_{\mathcal{L}})$ such that $\mathrm{U}(S(A)) = T(\mathrm{U}(A))$, and then prove that η^T and $(-)^T$ preserve relatedness at every level.

Exceptions

Programs that may throw exceptions can (ab)use termination as a channel to leak information about their input. These programs can either terminate without exceptions or with an exception code. An information-flow property for ideal security for these programs, which we may call "exception-sensitivity", disallows any leaks from the termination channel. A less ideal, yet practical, property allows information leaks through exceptional termination. We may call this "exception-insensitivity". This property is what type systems for information flow usually enforce—see e.g. Pottier and Simonet (2003) or Askarov and Sabelfeld (2009).

We model these security specifications by specification monads "living", in some sense, over Exc—see Section B.4.2. Unsurprisingly redaction at l and the specification monad corresponding to exception-sensitivity do not interact at any $l \in L$ other than $\bot$: throwing exceptions depending on redacted values leaks information and the monad specifies that this kind of leak is not permitted. On the other hand, for each level $l \in L$ there is a specification monad Exc_l^{TI} that corresponds to permitting leaks through termination at those levels $l' \not\sqsubseteq l$. Redaction at l and this monad interact provided that leaks are also admitted by exceptional termination with different exception codes.

In the rest of this section, let E be a set of exceptions.

Exception-sensitive specification monad We define a specification monad corresponding to exception-sensitivity using analogous type structure to that of the monad Exc but on $\mathrm{CSet}_{\mathcal{L}}$.

Recall that $\mathrm{CSet}_{\mathcal{L}}$ has finite coproducts cf. Theorem B.2.1. The coproduct of two classified sets may be explicitly constructed as follows:

Definition B.4.3 (cf. Kavvos (2019, Proposition 4))**.** Let A and B be classified sets, the coproduct of A and B is the classified set $A + B$ defined by

$$\begin{aligned} \mathrm{U}(A + B) &\coloneqq \mathrm{U}(A) + \mathrm{U}(B) \\ \mathrm{R}_l(A + B)(\mathrm{inl}(a_1), \mathrm{inl}(a_2)) &\coloneqq\Leftrightarrow \mathrm{R}_l(A)(a_1, a_2) \\ \mathrm{R}_l(A + B)(\mathrm{inr}(b_1), \mathrm{inr}(b_2)) &\coloneqq\Leftrightarrow \mathrm{R}_l(B)(b_1, b_2) \\ \mathrm{R}_l(A + B)(\mathrm{inl}(a), \mathrm{inr}(b)) &\coloneqq\Leftrightarrow \bot \end{aligned}$$

Further, we need a lifting of E to a classified set $\Delta(E)$. The type $\Delta(E)$ specifies that all observer can distinguish between all exceptions:

Definition B.4.4 (Discrete (cf. Kavvos (2019, Section 3)))**.** The functor $\Delta : \mathrm{Set} \to \mathrm{CSet}_{\mathcal{L}}$ maps a set X to the classified set

$$\mathrm{U}(\Delta(X)) \coloneqq X \qquad\qquad \mathrm{R}_l(\Delta(X))(x, y) \coloneqq\Leftrightarrow x = y$$

Any function $f : X \to Y$ trivially preserves indistinguishability at $\Delta(X) \to \Delta(Y)$.

The object mapping part of the exception-sensitive specification monad Exc^{TS} is given by

$$\mathrm{Exc}^{TS}(A) \coloneqq A + \Delta(E)$$

Since U preserves coproducts, Theorem B.2.2, and $\mathrm{U} \circ \Delta = \mathrm{Id}$ this object mapping lives over Exc. Hence by Remark B.4.4 to verify that this forms a monad it is enough to check that the operations η and $(-)^{\mathrm{Exc}}$ of Exc preserve relatedness at every level—which in fact they do. The operation for throwing exceptions is analogous to that on Set.

Interaction with $\blacklozenge_l$ For all $l \in L$ such that $l \neq \bot$, redaction at l and Exc^{TS} do not interact in the sense of Definition B.4.1. Intuitively this should be clear: the coproduct encodes an observable choice between left and right injections. Formally this is an instance of Remark B.3.2.

Exception-insensitive specification monad To define a specification monad corresponding to exception-insensitivity, as discussed in the beginning of this section, we need an alternative version of "coproducts".

The family of indistinguishability relations on the coproduct of two classified sets—see Definition B.4.3—relates two elements iff they belong to the same injection (and the underlying values are appropriately related). The last clause in Definition B.4.3 specifies that observers can distinguish between normal termination, i.e. left injection, and exceptional termination, i.e. right injection. This causes that the specification monad Exc^{TS} specifies termination-sensitivity. By weakening the relation on the coproduct, and hence admitting more elements being related, we obtain a new specification monad for exceptions that corresponds to (depending on a level l) termination-insensitivity. Formally this weakening corresponds to the following type constructor on $\mathrm{CSet}_{\mathcal{L}}$:

Definition B.4.5. Let A and B be classified sets and $l \in L$, define the classified set $A \oplus_l B$ by

$$\begin{aligned} \mathrm{U}(A \oplus_l B) &\coloneqq \mathrm{U}(A) + \mathrm{U}(B) \\ \mathrm{R}_{l'}(A \oplus_l B)(\mathrm{inl}(a_1), \mathrm{inl}(a_2)) &\coloneqq\Leftrightarrow \mathrm{R}_l(A)(a_1, a_2) \\ \mathrm{R}_{l'}(A \oplus_l B)(\mathrm{inr}(b_1), \mathrm{inr}(b_2)) &\coloneqq\Leftrightarrow \mathrm{R}_l(B)(b_1, b_2) \\ \mathrm{R}_{l'}(A \oplus_l B)(\mathrm{inl}(a), \mathrm{inr}(b)) &\coloneqq\Leftrightarrow l \not\sqsubseteq l' \end{aligned}$$

This definition amounts to say that at level l' two values of $\mathrm{U}(A) + \mathrm{U}(B)$ are related at $A \oplus_l B$ if either they are in the same injection (and related) or they are in different injections and $l \not\sqsubseteq l'$.

This type constructor supports several useful operations for programming:

Proposition B.4.2. *For any $l \in L$, $\oplus_l$ is a semicocartesian symmetric monoidal product. Explicitly, this means that*

1. $\oplus_l$ *extends to a bifunctor on* $\mathrm{CSet}_{\mathcal{L}}$.

2. $\oplus_l$ *is commutative and associative.*

3. $\oplus_l$ *supports injections* $\mathrm{inl}^l_{A,B} : A \to A \oplus_l B$ *and* $\mathrm{inr}^l_{A,B} B \to A \oplus_l B$ *(given by the standard set-theoretic injections on the underlying coproduct).*

4. *The initial object* 0 *in* $\mathrm{CSet}_{\mathcal{L}}$ *is unital w.r.t.* $\oplus_l$ *(This is equivalent to B.4.2.3).*

Unlike the coproduct, however,

Remark B.4.5. The type constructor $\oplus_l$ *does not*, in general, support copairing. In the two-point lattice $\mathtt{LH}$, consider the maps $t \coloneqq \lambda x.tt : 1 \to 2^{\mathtt{L}}$ and $f \coloneqq \lambda x.tt : 1 \to 2^{\mathtt{L}}$. Their copairing $[t, f]$ maps $\mathrm{inl}(\star)$ to tt and $\mathrm{inr}(\star)$ to tt. The type $1 \oplus_{\mathtt{H}} 1$ relates $\mathrm{inl}(\star)$ and $\mathrm{inr}(\star)$ at $\mathtt{L}$ but $\neg \mathrm{R}_{\mathtt{L}}(2^{\mathtt{L}})(tt, tt)$.

Remark B.4.6. For any security lattice $\mathcal{L} = (L, \sqsubseteq, \bot, \vee)$ the types $A + B$ and $A \oplus_\bot B$ are equal (by definition).

To complete the definition of the specification monad we need an exception-insensitive lifting of E, $\nabla(E)$. This type specifies that leaks through distinctions between exceptions are permitted at all levels $l \in L$. Formally,

Definition B.4.6 (Codiscrete (cf. Kavvos (2019, Section 3)))**.** The functor $\nabla : \mathrm{Set} \to \mathrm{CSet}_{\mathcal{L}}$ maps a set X to the classified set

$$\mathrm{U}(\nabla(X)) \coloneqq X \qquad\qquad \mathrm{R}_l(\nabla(X))(x, y) :\Leftrightarrow \top$$

Any function $f : X \to Y$ trivially preserves indistinguishability at $\nabla(X) \to \nabla(Y)$.

With these definitions in mind, we define a l-exception-insensitive specification monad by the following mapping on objects,

$$\mathrm{Exc}^{TI}_l(A) \coloneqq A \oplus_l \nabla(E)$$

The type constructor Exc^{TI}_l forms a monad. To see this note that the object mapping lives over Exc, i.e. $\mathrm{U}(A \oplus_l B) \doteq \mathrm{U}(A) + \mathrm{U}(B)$ and $\mathrm{U} \circ \nabla = \mathrm{Id}$, and the operations η and $(-)^{\mathrm{Exc}}$ of Exc preserve relatedness at every level as specified by $A \oplus_l \nabla(E)$.

Interaction with $\blacklozenge_l$ To understand the interaction between redaction at some level l and the monad Exc_l^{TI} we notice that being l-protected is closed under the type constructor $\oplus_l$:

Proposition B.4.3. *For all $l \in L$ and A, B classified sets over $\mathcal{L}$, if A and B are l-protected, then $A \oplus_l B$ is l-protected.*

Proof. We show that the identity function $\mathrm{id}_{\mathrm{U}(A)+\mathrm{U}(B)}$ is a map of classified sets $\blacklozenge_l(A \oplus_l B) \to A \oplus_l B$. The interesting case is for a level $l \not\sqsubseteq l'$ since $\mathrm{R}_{l'}(\blacklozenge_l(A \oplus_l B))(\mathrm{inl}(a), \mathrm{inr}(b))$ for any $a \in \mathrm{U}(A)$ and $b \in \mathrm{U}(B)$. By definition of $\oplus_l$, $\mathrm{R}_{l'}(A \oplus_l B)(\mathrm{inl}(a), \mathrm{inr}(b))$. □

In particular this implies that the two monads interact as per Theorem B.4.1. Intuitively, this says that it is secure to eliminate redacted values at l into computations since exception throwing cannot encode more information that what it is permitted to leak. We therefore conclude that the typing rule $\blacklozenge_l$-S-Elim is valid for l and Exc_l^{TI}.

Global State

Computations that read and modify the state of the machine (see Section B.4.2) can abuse side-effecting operations to leak information about their input or secret parts of the state. For example, a stateful computation may override some public part of the memory depending on data labelled as secret; this is the paradigmatic example of an *implicit flow* (Sabelfeld and Myers 2003, Section II.C).

Specification monads We define specification monads for stateful computations over a state represented by a classified set S using analogous type structure to that of St_S, i.e. exponentials and products (see Theorem B.2.1), but on $\mathrm{CSet}_{\mathcal{L}}$:

$$\mathrm{St}_S(A) \coloneqq S \Rightarrow A \times S$$

This type constructor forms a monad in the same way as the global state monad on Set (cf. Section B.4.2). Two closed computations, e.g. of type $A \to \mathrm{St}_S(B)$, are indistinguishable if they are extensionally indistinguishable: that is, they map related inputs of type $A \times S$ to related outputs of type $B \times S$. Note that, like the monad on Set, the operations of reading and writing are transparent.

Remark B.4.7. The specification monad St_S, for a state S, does not, in general, "live" over the global state monad $\mathrm{St}_{\mathrm{U}(S)}$ on Set, i.e. $\mathrm{U}(\mathrm{St}_S(A)) \doteq \mathrm{U}(S \Rightarrow A \times S) \neq \mathrm{U}(S) \Rightarrow \mathrm{U}(A) \times \mathrm{U}(S)$. Intuitively, the exponential object $S \Rightarrow$

$A \times S \coloneqq \mathrm{Hom}_{\mathrm{CSet}_{\mathcal{L}}}(S, A \times S)$ in classified sets contains only those programs of type $\mathrm{U}(S) \Rightarrow \mathrm{U}(A) \times \mathrm{U}(S)$ that are noninterferent.

Example B.4.1. In the two-point lattice $\mathtt{LH}$, consider the monad $\mathrm{St}_{2^{\mathtt{L}} \times 2^{\mathtt{H}}}$. This type constructor models side-effects over a store that consists of a public and a secret Boolean. Computations of type $2^{\mathtt{H}} \to \mathrm{St}_{2^{\mathtt{L}} \times 2^{\mathtt{H}}}(1)$ may modify the input store depending on the secret Boolean. After execution, the secret part of the store, i.e. of type $2^{\mathtt{H}}$, may depend on the secret Boolean and both the secret and public parts of the input store; on the other hand, its public part can only depend on the public part of the input store.

Remark B.4.8. Type systems for information flow typically use a *program counter* (pc) label to keep track of the control-flow and restrict what parts of the state computations can be modified to prevent implicit flows—see e.g. S. A. Zdancewic (2002, Section 3.2). In hindsight, this amounts to writing computations in a monad other than state. For example in the two-point lattice, a computation that has access to a read-only public Boolean, i.e. of type $2^{\mathtt{L}}$, and that computes over a secret store of type Boolean corresponds to the monad $A \mapsto 2^{\mathtt{L}} \Rightarrow (2^{\mathtt{H}} \Rightarrow A \times 2^{\mathtt{H}})$ (the $2^{\mathtt{L}}$-reader monad stacked over the $2^{\mathtt{H}}$-state monad). There is a monad morphism from this monad to $\mathrm{St}_{2^{\mathtt{L}} \times 2^{\mathtt{H}}}$, and hence, it can be though of as a restricted version of the latter: in this "state monad", the put operation only writes to the secret part of the state.

Interaction with $\blacklozenge_l$ The interaction of the monad $\blacklozenge_l$ and St_S for some state S depends on whether S is l-protected. To see this, assume S is l-protected; then for all $A \in \mathrm{Obj}(\mathrm{PSet}^l_{\mathcal{L}})$ it is the case that $\mathrm{St}_S(A)$ is l-protected. This follows from Propositions B.3.4.2 and B.3.4.3. We therefore conclude that the typing rule $\blacklozenge_l$-S-Elim is valid for stateful computations, as specified by a state monad St_S, whenever the state S is adequately protected.

Nondeterminism

Programs with nondeterministic choice can leak information by varying the set of possible outcomes of a computation depending on secret data. If the type of these possible outcomes is suitably protected, however, then the amount of information conveyed to observers is limited.

Specification monads We define a specification monad for nondeterministic computations by the following mapping on objects:

$$\begin{aligned} \mathrm{U}(\mathrm{NDet}(A)) &\coloneqq \mathrm{P}^+(\mathrm{U}(A)) \\ \mathrm{R}_l(\mathrm{NDet}(A))(A_0, A_1) &:\Leftrightarrow (\forall a_0 \in A_0.\, \exists a_1 \in A_1 \wedge \mathrm{R}_l(A)(a_0, a_1)) \\ &\quad \wedge (\forall a_1 \in A_1.\, \exists a_0 \in A_0 \wedge \mathrm{R}_l(A)(a_1, a_0)) \end{aligned}$$

This type construct forms a monad: NDet lives over NDet, i.e. $\mathrm{U}(\mathrm{NDet}(A)) \doteq \mathrm{P}^+(\mathrm{U}(A))$, and the operations η and $(-)^{\mathrm{NDet}}$ of NDet (in Set) preserve relatedness at every level. Note that the relation on this type relates two (nonempty finite) subsets with elements of type $\mathrm{U}(A)$ whenever for all elements in one subset there is a related element in the other subset, and vice versa.

Example B.4.2. Consider the nondeterministic program:

$$\begin{aligned} &p : 2 \to \mathrm{NDet}(2) \\ &p \coloneqq \lambda b.\, \text{if } b \text{ then return}(\{tt\}) \text{ else return}(\{tt, tt\}) \end{aligned}$$

where the set-builder notation means nondeterministic choice amongst the given elements.

In the two-point lattice $\mathtt{L}\mathtt{H}$, this program is a map of classified sets $2^{\mathtt{H}} \to \mathrm{NDet}(2^{\mathtt{H}})$: observers at level $\mathtt{L}$ can not distinguish between the subsets $\{tt\}$ and $\{tt, tt\}$. On the other hand, this program is not a map of classified sets $2^{\mathtt{H}} \to \mathrm{NDet}(2^{\mathtt{L}})$: in one execution the program may return tt, while in another it may return tt depending on the secret input, and these values can be distinguished.

Remark B.4.9. This specification monad is an adaptation of the construction in Sabelfeld and Sands (2001, Section 3.3) to lift partial equivalence relations on domains, which they use to model indistinguishability, to the various notions of powerdomains, i.e. Hoare, Smyth, and Plotkin's. In contrast to powerdomains, the semantics of nondeterminism in Set, as given by the P^+ monad, does not model failure. For that we would need a different monad such as, e.g., the finite powerset monad.

Interaction with $\blacklozenge_l$ The monads $\blacklozenge_l$ and NDet interact. The proof amounts to show that for any l-protected classified set A, $\mathrm{id}_{\mathrm{P}^+(\mathrm{U}(A))} : \blacklozenge_l(\mathrm{NDet}(A)) \to \mathrm{NDet}(A)$.

Proof sketch. Consider the case for a level $l' \in L$ such that $l \not\sqsubseteq l'$. Recall that for all $a_0, a_1 \in \mathrm{U}(A)$, $\mathrm{R}_{l'}(\blacklozenge_l(A))(a_0, a_1)$—see Definition B.3.1. For all nonempty

finite subsets $A_0, A_1 \in \mathrm{P}^+(\mathrm{U}(A))$ we have to show that $\mathrm{R}_{l'}(\mathrm{NDet}(A))(A_0, A_1)$, For this, we show that $\forall a_0 \in A_0.\, \exists a_1 \in A_1 \wedge \mathrm{R}_{l'}(A)(a_0, a_1)$. The other part of the conjunct follows a similar argument.

By assumption A_1 is nonempty, and thus, for all $a_0 \in A_0$ there is an element $a_1 \in A_1$. By the definition of the relation $\mathrm{R}_{l'}(\blacklozenge_l(A))$ it is the case that $\mathrm{R}_{l'}(a_0, a_1)$, and thus, we obtain. □

We conclude that the typing rule $\blacklozenge_l$-S-Elim is valid for this specification monad.

Interactive Input–Output

Programs that interactively consume input and/or produce output can (ab)use several channels to leak information. To name a couple of these channels: a program can consume its input any number of times depending on previous input, which may be secret, or initial secrets; or, a program may encode the contents of a secret value in the number of times that the program prints (irrespectively of the actual data that it prints). These forms of leakage arise by publicly observable differences in the structure, i.e. the "trees" (see Section B.4.2), that represents $\mathrm{IO}_{I,O}$ computations, when executed with distinct secret inputs. Recently, Silver et al. (2023) explored a similar approach using interaction trees.

We model several security specifications for interactive programs by means of specification monads living over $\mathrm{IO}_{\mathrm{U}(I),\mathrm{U}(O)}$, and parameterised by classified sets for the input and output types, i.e., I and O, respectively. These security specifications range from ideal security, where no leakage is permitted at all: only computations that proceed in tight lockstep are deemed indistinguishable to all observers; to less ideal notions where, for example, it is acceptable that indistinguishable computations consume inputs a different number of times, or produce a different number of outputs (for example, when certain observers do not have access to particular output channels). On the extreme side there is a security specification that permits any two computations to differ in all inputs and outputs. Unsurprisingly, redaction at l only interacts with the monad corresponding to this last specification.

Let $I, O \in \mathrm{Obj}(\mathrm{CSet}_{\mathcal{L}})$ in the rest of this section.

Specification monads for ideal security We define a specification monad that relates two computations only when they consume and produce output in synchrony with each other. This models a threat model where both producing output and consuming input, seen as a "side-effect", is publicly observable (e.g.

using side-channel attacks), and hence, these differences may leak information about the secret inputs.

This specification monad is defined by the mapping on objects,

$$\mathrm{U}(\mathrm{IO}^S_{I,O}(A)) \coloneqq \mathrm{IO}_{\mathrm{U}(I),\mathrm{U}(O)}(\mathrm{U}(A))$$

together with the following family of relations,

Definition B.4.7. Let $l \in L$, we define an inductive relation $\mathrm{R}_l(\mathrm{IO}_{I,O}(A))$ on $\mathrm{IO}_{\mathrm{U}(I),\mathrm{U}(O)}(\mathrm{U}(A)))$ by the following rules:

$$\textsc{Return}\quad \frac{\mathrm{R}_l(A)(a_1,a_2)}{\mathrm{R}_l(\mathrm{IO}^S_{I,O}(A))(\mathrm{return}(a_1),\mathrm{return}(a_2))}$$

$$\textsc{Out}\quad \frac{\mathrm{R}_l(O)(o_1,o_2) \qquad \mathrm{R}_l(\mathrm{IO}^S_{I,O}(A))(k_1,k_2)}{\mathrm{R}_l(\mathrm{IO}^S_{I,O}(A))(\mathrm{out}(o_1,k_1),\mathrm{out}(o_2,k_2))}$$

$$\textsc{In}\quad \frac{\forall i_1,i_2 \in \mathrm{U}(I).\, \mathrm{R}_l(I)(i_1,i_2) \Rightarrow \mathrm{R}_l(\mathrm{IO}^S_{I,O}(A))(k_1(i_1),k_2(i_2))}{\mathrm{R}_l(\mathrm{IO}^S_{I,O}(A))(\mathrm{in}(k_1),\mathrm{in}(k_2))}$$

We briefly explain these inductive rules. At level $l \in L$, rule Return relates two leaves, whenever the values they contain are likewise related; rule Out relates two computations that produce outputs, if the outputs themselves are related and the rest of the computations are also related; lastly, rule In relates two computations that depend on inputs of type $\mathrm{U}(I)$, whenever supplying related inputs at type I yields related computations. This type constructor forms a monad on $\mathrm{CSet}_{\mathcal{L}}$: the operations η and $(-)^{\mathrm{IO}}$ yield maps of the appropriate type in classified sets. Note that this monad lives directly $\mathrm{IO}_{\mathrm{U}(I),\mathrm{U}(O)}$.

Example B.4.3. Consider the computational monad $\mathrm{IO}_{2,2}$, that is, for interactive programs that receive inputs of unit type and produce outputs of type 2, and the program

$$p : 2 \to \mathrm{IO}_{2,2}(2)$$
$$p \coloneqq \lambda b.\, \mathrm{if}\ b\ \mathrm{then}\ \mathrm{in}(\lambda u'.\mathrm{return}(tt))\ \mathrm{else}\ \mathrm{return}(tt)$$

In the two-point lattice $\mathtt{LH}$, this function is *not* a map of classified sets $2^{\mathtt{H}} \to \mathrm{IO}^S_{1,2^{\mathtt{L}}}(2^{\mathtt{L}})$. Depending on the (secret) input b the program p either reads one or zero times and this difference constitutes a forbidden leak, i.e. $\neg\mathrm{R}_{\mathtt{L}}(\mathrm{IO}^S_{1,2^{\mathtt{L}}}(2^{\mathtt{L}}))(p(tt),p(tt))$.

Interaction with $\blacklozenge_l$ The monads $\blacklozenge_l$ and $\mathrm{IO}^S_{I,O}$ do not interact. In the two-point lattice $\mathtt{LH}$, consider the type $\mathrm{IO}^S_{1,1}(1)$, and recall that 1 is $\mathtt{H}$-protected (Proposition B.3.4.1). Then, $\mathrm{R}_{\mathtt{L}}(\blacklozenge_{\mathtt{H}}(\mathrm{IO}^S_{1,1}(1)))(\mathrm{out}(\mathrm{return}(\star)), \mathrm{return}(\star))$, however, it is not the case that these computations are also related at type $\mathrm{IO}^S_{1,1}(1)$, i.e. $\neg \mathrm{R}_{\mathtt{L}}(\mathrm{IO}^S_{1,1}(1))(\mathrm{out}(\mathrm{return}(\star)), \mathrm{return}(\star))$.

Other specification monads The security property encoded by the specification monads as constructed in the previous section can be weakened along the input and output axes to permit certain degree of leakage from differences in the structure of computations. In particular, these "weaker" specification monads allow leaks by relating computations that may differ on the inputs and outputs they produce.

We define several specification monads whose mapping on objects is given, as above, by the following:

$$\mathrm{U}(\mathrm{IO}_{I,O}(A)) \coloneqq \mathrm{IO}_{\mathrm{U}(I),\mathrm{U}(O)}(\mathrm{U}(A))$$

The family of relations of these monads is based on the family above, i.e. in Definition B.4.7, except that we modify some clauses according to what part of the security specification we wish to weaken:

1. To allow leakage through the input channel, that is, to permit computations that differ in the number of consumed inputs, we replace the clause In by the following four rules:

$$\frac{\forall i \in \mathrm{U}(I).\, \mathrm{R}_l(\mathrm{IO}_{I,O}(A))(k(i), \mathrm{out}(o, k'))}{\mathrm{R}_l(\mathrm{IO}_{I,O}(A))(\mathrm{in}(k), \mathrm{out}(o, k'))}$$

$$\frac{\forall i \in \mathrm{U}(I).\, \mathrm{R}_l(\mathrm{IO}_{I,O}(A))(k(i), \mathrm{return}(a))}{\mathrm{R}_l(\mathrm{IO}_{I,O}(A))(\mathrm{in}(k), \mathrm{return}(a))}$$

$$\frac{\forall i \in \mathrm{U}(I).\, \mathrm{R}_l(\mathrm{IO}_{I,O}(A))(\mathrm{out}(o, k'), k(i))}{\mathrm{R}_l(\mathrm{IO}_{I,O}(A))(\mathrm{out}(o, k'), \mathrm{in}(k))}$$

$$\frac{\forall i \in \mathrm{U}(I).\, \mathrm{R}_l(\mathrm{IO}_{I,O}(A))(\mathrm{return}(a), k(i))}{\mathrm{R}_l(\mathrm{IO}_{I,O}(A))(\mathrm{return}(a), \mathrm{in}(k))}$$

At a level $l \in L$, this family of relations relates any two computations that synchronize when they produce outputs, see rule Out, and/or return values (rule Return), but not on consuming inputs. This type constructor forms a monad on $\mathrm{CSet}_{\mathcal{L}}$.

Interaction with $\blacklozenge_l$ The monads $\blacklozenge_l$ and $\mathrm{IO}_{I,O}$ do not interact: the counterexample in Section B.4.3 still applies.

2. To permit leakage through the output channel, we replace the clause Out in Definition B.4.7 by:

$$\frac{\mathrm{R}_l(\mathrm{IO}_{I,O}(A))(\mathrm{in}(k), k')}{\mathrm{R}_l(\mathrm{IO}_{I,O}(A))(\mathrm{in}(k), \mathrm{out}(o, k'))} \qquad \frac{\mathrm{R}_l(\mathrm{IO}_{I,O}(A))(k, \mathrm{return}(a))}{\mathrm{R}_l(\mathrm{IO}_{I,O}(A))(\mathrm{out}(o, k), \mathrm{return}(a))}$$

$$\frac{\mathrm{R}_l(\mathrm{IO}_{I,O}(A))(k', \mathrm{in}(k))}{\mathrm{R}_l(\mathrm{IO}_{I,O}(A))(\mathrm{out}(o, k'), \mathrm{in}(k))} \qquad \frac{\mathrm{R}_l(\mathrm{IO}_{I,O}(A))(\mathrm{return}(a), k)}{\mathrm{R}_l(\mathrm{IO}_{I,O}(A))(\mathrm{return}(a), \mathrm{out}(o, k))}$$

At level $l \in L$, this family of relations relates two computations if whenever they demand inputs and/or return values they do so in a lockstep fashion (see clauses In and Return, respectively. These computations can, however, produce different number of outputs and these need not to be related. This type constructor again forms a monad on $\mathrm{CSet}_{\mathcal{L}}$.

Interaction with $\blacklozenge_l$ The monads $\blacklozenge_l$ and $\mathrm{IO}_{I,O}$ do not interact. In the two-point lattice, take two closed computations, $\mathrm{in}(\lambda u.\mathrm{return}(\star))$ and $\mathrm{return}(\star)$, at type $\mathrm{IO}_{1,1}(1)$. These computations are not related at level L, however, they are related at L when considered at type $\blacklozenge_{\mathrm{H}}(\mathrm{IO}_{1,1}(1))$.

3. To allow leakage both through the input and output axes, combine the clauses in 1 and 2. At level $l \in L$, the family of relations relates any two computations as long as in the end they return related values at l.

Interaction with $\blacklozenge_l$ The monads $\blacklozenge_l$ and $\mathrm{IO}_{I,O}$, as defined by this relation, interact.

Proof sketch. Note that for any $l \in L$, if $\mathrm{R}_l(A)$ is the everywhere true relation, i.e. for all $a_1, a_2 \in \mathrm{U}(A)$, $\mathrm{R}_l(A)(a_0, a_1)$, then $\mathrm{R}_l(\mathrm{IO}_{I,O}(A)$ is also everywhere true. Consider the case $l' \not\sqsubseteq l$, where the relation $\mathrm{R}_{l'}(A)$ is everywhere true—see Definition B.3.1. Therefore, $\mathrm{R}_{l'}(\mathrm{IO}_{I,O}(A)$ is also everywhere true. □

Hence, we conclude that the typing rule $\blacklozenge_l$-S-Elim is valid for this monad: the effects of consuming input and/or producing output depending on does not leak more information than what is permitted.

Remark B.4.10. Note that the specification monads we have defined for interactive input–output computations can immediately be abstracted into a family specification monads over $\mathrm{IO}_{\mathrm{U}(I),\mathrm{U}(O)}$ indexed by pairs of levels $l_i, l_o \in L$, where these levels represent the security boundaries on which leakage through the *i*nput and/or *o*utput channels is allowed.

B.5. Nontermination

A study of information flow with effects would not be complete if it did not include a treatment of divergence. Information-flow properties of nonterminating programs are typically modelled in categories of (pre)domains and suitable relations, see e.g. Abadi et al. (1999) or Hunt et al. (2023). However, in this work we take a more ergonomic approach by using Capretta (2005) delay monad on the category of sets.[2] This has the advantage of fitting in the same framework as the other effects studied in Section B.4.2, namely, categories of classified sets.

Recall the definition of the delay monad: Capretta (2005, Definition 3.1)

Definition B.5.1. Let X be a set, we define the coinductive type $\mathrm{D}(X)$, pronounced delay, by the following rules:[3]

$$\frac{x : X}{\mathrm{now}(x) : \mathrm{D}(X)} \qquad \frac{xs : \mathrm{D}(X)}{\mathrm{later}(xs) : \mathrm{D}(X)}$$

This type constructor forms a computational monad on Set. The operation $\eta : X \to \mathrm{D}(X)$ returns a value via the constructor now; and, sequencing takes a computation $f : X \to \mathrm{D}(Y)$ to a computation $f^{\mathrm{D}} : \mathrm{D}(X) \to \mathrm{D}(Y)$ that applies the function f to an element $\mathrm{D}(X)$ whenever this is defined, i.e. it consists of a finite number of constructors later followed by $\mathrm{now}(x)$ for some $x : X$. The operation for producing effects, that is, divergence, is $\mathrm{never} \coloneqq \mathrm{later}(\mathrm{never}) : 1 \to \mathrm{D}(0)$. This closed program is an example of a definition by coinduction.

B.5.1. Security and Specification Monads

Programs that may fail to terminate can (ab)use the so-called termination channel to leak information about their secret input. An information-flow property for ideal security, usually known as termination-sensitive noninterference,

[2] This further aligns with the fact that we work within a constructive/type theoretic metatheory.

[3] Double lines signify coinductive rules, and single lines inductive rules.

forbids information leakage through this channel. On the other hand, a more liberal property, namely termination-insensitive noninterference, does allow information leaks arising from this channel. The latter is what type systems for information flow usually encode. In the presence of divergence together with other effects, e.g. such as printing for example, this property is inadequate—see Askarov, Hunt, et al. (2008).

We model these properties using specification monads "living" over the delay monad.

Termination-sensitive specification monad We define a specification monad corresponding to termination-*S*ensitivity by the mapping on objects,

$$\mathrm{D}^S(A) \coloneqq \mathrm{D}(\mathrm{U}(A))$$

together with the following family of relations,

Definition B.5.2. Let $l \in L$, we define a relation $\mathrm{R}_l(\mathrm{D}^I(A))$ on the set $\mathrm{D}(\mathrm{U}(A))$ by the following rules:

$$\textsc{Now}\ \frac{\mathrm{R}_l(A)(x,y)}{\mathrm{R}_l(\mathrm{D}^S(A))(\mathrm{now}(x),\mathrm{now}(y))} \qquad \textsc{Later}\ \frac{\mathrm{R}_l(\mathrm{D}^S(A))(xs,ys)}{\mathrm{R}_l(\mathrm{D}^S(A))(\mathrm{later}(xs),\mathrm{later}(ys))}$$

$$\textsc{Later-Now}\ \frac{\mathrm{R}_l(\mathrm{D}^S(A))(xs,\mathrm{now}(y))}{\mathrm{R}_l(\mathrm{D}^S(A))(\mathrm{later}(xs),\mathrm{now}(y))} \qquad \textsc{Now-Later}\ \frac{\mathrm{R}_l(\mathrm{D}^S(A))(\mathrm{now}(x),ys)}{\mathrm{R}_l(\mathrm{D}^S(A))(\mathrm{now}(x),\mathrm{later}(ys))}$$

Note that this definition uses a mix of inductive and coinductive rules.

The relation $\mathrm{R}_l(\mathrm{D}^I(A))$ relates two elements of type $\mathrm{D}(\mathrm{U}(A))$ if they are either both undefined, i.e. equal to never, or both defined with a possibly different number of "steps", and their underlying values are further related by $\mathrm{R}_l(A)$. As expected, this type constructor forms a monad on $\mathrm{CSet}_{\mathcal{L}}$.

Remark B.5.1. This relation (at some $l \in L$) is usually known as weak bisimilarity, see, e.g., Chapman et al. (2019, Section 3).

Interaction with $\blacklozenge_l$ The monads $\blacklozenge_l$ and D^S do not interact. In the two-point lattice LH, consider the type $\mathrm{D}^S(1)$. and recall that the type 1 is H-protected—see Proposition B.3.4.1. Then the programs never $\coloneqq$ later(never) and now($\star$) are related at type $\blacklozenge_{\mathtt{H}}(\mathrm{D}^S(1))$ at level L, that is, $\mathrm{R}_{\mathtt{L}}(\blacklozenge_{\mathtt{H}}(\mathrm{D}^S(1)))(\mathrm{never},\mathrm{now}(\star))$,

however, these programs are not related at type $\mathrm{D}^S(1)$ and at level $\mathtt{L}$, that is, $\neg \mathrm{R}_{\mathtt{L}}(\mathrm{D}^S(1))(\mathrm{never}, \mathrm{now}(\star))$.

We therefore conclude that the typing rule $\blacklozenge_l$-S-Elim for this monad is not valid.

Termination-insensitive specification monad To define a specification monad that corresponds to termination-*I*nsensitivity, we use the same mapping on objects as above,

$$\mathrm{D}^I(A) \coloneqq \mathrm{D}(\mathrm{U}(A))$$

together with the family of relations:

Definition B.5.3. Let $l \in L$, we define a coinductive relation $\mathrm{R}_l(\mathrm{D}^I(A))$ on the set $\mathrm{D}(\mathrm{U}(A))$ by replacing Later-Now and Now-Later by the following rules:

$$\frac{\mathrm{R}_l(\mathrm{D}^I(A))(xs, \mathrm{now}(y))}{\mathrm{R}_l(\mathrm{D}^I(A))(\mathrm{later}(xs), \mathrm{now}(y))} \qquad \frac{\mathrm{R}_l(\mathrm{D}^I(A))(\mathrm{now}(x), ys)}{\mathrm{R}_l(\mathrm{D}^I(A))(\mathrm{now}(x), \mathrm{later}(ys))}$$

At level l, this relation relates two values of type $\mathrm{D}(\mathrm{U}(A))$ if either one of them is undefined, i.e. equal to never, or both are defined and, regardless of the number of steps, their underlying values are related by $\mathrm{R}_l(A)$. This type constructor forms a monad on $\mathrm{CSet}_{\mathcal{L}}$.

Remark B.5.2. In the two-point lattice, the relation at level $\mathtt{L}$ at type $\mathrm{D}^I(2^{\mathtt{L}})$, i.e. $\mathrm{R}_{\mathtt{L}}(\mathrm{D}^I(2^{\mathtt{L}})$, is reflexive and symmetric but crucially not transitive. The lack of transitivity is what permits it to relate, e.g., never and any other value of type $\mathrm{D}(2)$ without trivializing the relation.

Example B.5.1. Consider the program $\lambda b.\,\mathrm{if}\ b\ \mathrm{then\ never\ else\ now}(\star)$, which depending on the input Boolean diverges or just returns unit. In the two the two-point lattice $\mathtt{LH}$ this program is a map of classified sets $2^{\mathtt{H}} \to \mathrm{D}^I(1)$ since $\mathrm{R}_{\mathtt{L}}(\mathrm{D}^I(1))(\mathrm{never}, \mathrm{now}(\star))$.

Interaction with $\blacklozenge_l$ The monads $\blacklozenge_l$ and D^I interact:

Proof sketch. Let A be a $\mathcal{L}$-classified set. First note that for all $l \in L$, the relation $\mathrm{R}_l(\mathrm{D}^I(A))$ is everywhere true whenever the relation $\mathrm{R}_l(A)$ is.

Consider two levels $l \not\sqsubseteq l' \in L$. Then, by definition (Definition B.3.1), the relation $\mathrm{R}_{l'}(A)$ is everywhere true, and thus, so is $\mathrm{R}_{l'}(\mathrm{D}^I(A))$. □

We hence conclude that the typing rule $\blacklozenge_l$-S-Elim is valid for this monad.

B.6. Related Work

Kavvos (2019) presents a family of models of information flow for higher-order programs without effects, modelled as set-theoretic functions. These models, i.e., categories of classified sets, are the starting point of our study. In contrast to Kavvos, we are concerned with information-flow properties for programs *with general effects*. We give semantics to effects, and their security specifications using computational monads in these categories.

Kavvos' work was partly inspired by Abadi et al.'s models of information flow for higher-order programs with divergence (Abadi et al. 1999), where programs are modelled by domain-theoretic continuous functions. In these models, based on suitable families of relations and relation preserving functions, they construct two so-called "lifting" monads that correspond to termination-sensitive and termination-insensitive security specifications—see Abadi et al. (1999, Sections 3.2 and 5). This can be seen as an instance of our work in the case of divergence. Further, they notice the natural isomorphism that distributes redaction at any level over the termination-insensitive lifting monad. In this sense, our work is a vast generalization of this observation to other effects. In addition we make precise this observation and study under what conditions, for any effect given by a monad, this kind of interaction exists.

Hunt et al. (2023) present similar but more refined models to those of Abadi et al. (1999). These models restrict the family of relations to those that arise as the generalized kernel of some continuous function into a "domain of observations". This formalizes the intuitive idea that some domain elements contain *qualitatively* better information than others, based on their degree of definedness. Having a handle on this, they show how to generalize termination-insensitive properties to domains with nontrivial depth (i.e. not only flat domains). However, their construction leaves the realm of these restricted relations, and thus, is not clear yet what is the appropriate universe of types on which it forms a monad, if any. Nonetheless, they apply their construction to the case of nondeterminism given by using Plotkin's powerdomain on the lifted Booleans and obtain a sensible security specification for programs. However, they do not consider how this construction interacts with redaction. In contrast, we consider nondeterminism in Set, other kinds of effects, and how these might interact with redaction.

Sabelfeld and Sands (2001) study information flow for nondeterministic and possibly divergent first-order programs—represented as continuous functions from states to powerdomains of states. They consider several variations of powerdomains for giving semantics to nondeterministic programs—i.e. Hoare, Smyth, and Plotkin's—and study the security properties obtained by (relation-

ally) lifting partial equivalence relations from the set of states. The specification monad for nondeterminism that we define is based on their approach. In contrast to their work, we are interested in information flow for higher-order programs, and consider effects in general.

Common to all these models, and hence ours, is that they follow what we may call a relational approach to information flow. The origin of this approach can be traced back to Landauer and Redmond (1993).

Sterling and Harper (2022) propose a different approach using models of information flow based on presheaves of domains instead of relations. They focus concretely on models for higher-order programs with only termination-insensitivity divergence: their lifting monad is automatically termination-insensitive. They show that redaction (at any level) interacts with this monad, and realize this in a calculus, that resembles CbPV and DCC, by a primitive to "declassify" the termination channel. This primitive, which is syntactically similar to the typing rule $\blacklozenge_l$-S-Distr-Elim, declassifies the termination behaviour of programs according to the security level of each observer. In contrast, distributive laws as presented here treat monads for effects and redaction in a uniform fashion.

Abramsky and Jagadeesan (2009) present yet another different kind of models of information flow based on an extension of game semantics in the style of AJM (Abramsky, Malacaria, et al. 1994). In contrast to this paper, their models concern higher-order programs without effects.

The works we already mentioned have, like ours, a very semantic flavour; we now move to discuss models of information flow that are not to be thought of as just giving semantics, in the broad sense of *meaning*, to programs, but as tools for proving security properties for information-flow type systems and calculi.

Tse and S. Zdancewic (2004) attempted to prove noninterference for DCC by a (sound and complete) translation to System F. Their motivation is an apparent connection between the relational models of Abadi et al. (1999) and Reynolds (1983) relational semantics for parametricity in System F. Sadly, it turned out that their translation was not fully abstract (Shikuma and Igarashi 2008), and thus, parametricity of System F did not imply noninterference of DCC. Bowman and Ahmed (2015) later resolved this issue by modifying the translation. The connection between parametricity and information flow has been further explored by Algehed and Bernardy (2019). In contrast to the present work, the relation between parametricity and information-flow properties has only been studied in the context of languages without effects. It does not seem a trivial task to extend this connection to languages with computational effects—although parametricity for the latter has been studied, e.g., by Møgelberg and Simpson (2007).

The pioneering work by Heintze and Riecke (1998) used a logical relations model constructed on top of a denotational semantics to prove noninterference for the SLam calculus, which is a higher-order language with divergence. They mention that their approach borrows ideas directly from the work by Reynolds on parametricity (Reynolds 1983). More recently, Gregersen et al. (2021) present a (mechanized) logical relations model for proving noninterference for a realistic language with many complicated and impressive features, such as recursive and (impredicative) polymorphic types, or higher-order state. Unlike the present work, these models aim to be used to give proofs of noninterference for the concrete languages they study, but not with providing a general semantic explanation of the mixture of effects, labelling, and information flow.

Modalities for information flow are a recurring example in the literature on coeffect systems, and hence, it deserves some discussion.

Gaboardi et al. (2016) present a calculus for mixing effects and coeffects via distributive laws. They show that the combination of information flow and nondeterminism forms an instance of their calculus via the construction of categories of information flow based on families of sets indexed by a security lattice. Similar to our work, they model effects by monads. However, they did not recognise that monads on categories of information flow encode security properties for effectful programs. Further, they did not observe that it is a property of redaction and monads for effects whether there exists a distributive law. These are key contributions of the present work.

Abel and Bernardy (2020) present a coeffect calculus together with a relational semantics inspired by parametricity and classified sets. Their calculus and semantics gives a unified view on modalities and subsumes previous work in the area. In contrast to our work, they do not consider computational effects and how these might interact with redaction as a coeffect.

Lago, Gavazzo, and Levy (2017) introduce a semantic framework that unifies several well-known theories of program equivalence for higher-order languages with effects. In their framework, effects are modelled by certain monads on the category of sets and "effect observations" by *relators* over these monads. Recently, Lago and Gavazzo (2022) extended this work to account for coeffects via the novel concept of *corelators*. They propose sufficient conditions under which a corelator and a relator interact nicely, namely that the former distributes over the latter. In contrast to the present paper, the focus of the aforementioned line of work is the semantics of calculi for program equivalence. This semantics requires that relations form a congruence over the constructs of programs, which, in particular, means that these need to be transitive. As previously discussed, transitivity is not desired, neither necessary, in the encoding of security properties that permit certain degree of leakage, e.g. termination-

insensitivity. Therefore, it is not clear how to encode certain security properties in their setting. This requires further investigation.

In recent work, Hirsch and Cecchetti (2021) present an axiomatic framework for, among other things, proving noninterference properties for languages with effects. Their key insight is that noninterference for an effectful language follows from noninterference for a pure language together with an appropriate translation from the former to the latter. This translation requires that effects are encodable in the pure language by a family of type constructors, which they model after effectors (Tate 2013). Their translation resembles that of monadic translations of type-and-effect systems (Wadler and Thiemann 2003). Further, they utilize this framework to formalize the folklore that *pc labels* serve as a lower bound on effects. Our work can be understood as substantiating Hirsch and Cecchetti's axiomatic framework: a category of classified sets forms a model of their pure language, and monads on such categories give semantics to type constructors for effects. In this sense, their approach could be classified as axiomatic while ours is denotational. In contrast to their work, ours gives a proper semantic justification to the existence of nice interactions of labelling and monads for effects in the form of (necessarily unique) distributive laws of redaction over these monads.

B.7. Conclusions and Further Work

In this paper, we have shown that monads on models of information flow, particularly on categories of classified sets, are suitable abstractions for encoding information-flow properties, and security specifications, of programs with general effects. Moreover, we have shown that soundly extending the usual elimination principles for "labelling", which pervasively appears in type systems and semantic models of information flow, to these effects corresponds to the existence of a distributive law from the redaction monad at some security level and the monad for effects. In fact, we have demonstrated, that whenever this distributive law exists it is necessarily unique, and thus, we can affirm, that once effects, and their security specifications are given by a monad, then it is a property of redaction and this monad whether it is secure to eliminate redacted values to computations producing effects. This novel observation is not particular to the model of classified sets but it is broadly applicable in other well-known models of information flow, since it depends on abstract properties of labelling and monads. We hope that this novel perspective brings a renewed interest on denotational models for security, information flow, and effects.

Before concluding, we discuss some possible venues for further work:

Categories of Domains In this work, we have confined our effects to monads on categories of classified sets, which are based on programs as set theoretic functions. Typically, denotational semantics of programs—á la Scott—are given in categories of domains (dcpo, ω-cpo, e.g.). Categories of information flow which take these as a basis already exists (e.g. Hunt et al. (2023), but only few effects (i.e. nontermination and nondeterminism) have been studied, and not from the perspective in this work. It would be interesting to do so.

Other Effects We have shown that the classical effects of Moggi (with exception of continuations) can be recasted as monads on categories of classified sets. In this regard we have scratched only the surface. A further work would be to tackle combinations of these effects and other effects. Moreover, it has been show that the appropriate semantic framework for higher-order store (Levy 2002) or name generation (Pitts and Stark 1993) as an effect are monads on functor categories. It would be interesting to extend the theory to information-flow properties of programs in these models.

Models of Effects In this paper, we have presented specification monads as strong monads on categories of classified sets. Since Moggi first proposed to used monads for modelling effects (Moggi 1989), an extensive literature has been developed looking for other structures; for example, graded monads (Katsumata 2014), parameterised monads (Atkey 2009) or effectors (Tate 2013). It would be interesting to 1. use these structures on categories of information flow to model "refined" notions of effects, and their refined security specifications, and 2. to further develop the theory of interaction between redaction and these structures.

Semantic Proofs of Information-Flow Properties We have presented a framework for modular reasoning about information-flow properties of programs with general effects. In its current form it is mainly a semantic tool. However, it would be interesting to use the theory here developed to establish these properties for type systems for information flow with effects.

Acknowledgements

I would like to thank Alejandro Russo, Sebastian Hunt, Fabian Ruch, and the anonymous referees for their feedback on earlier versions of this work. I would also like to thank David Sands for our discussions.

This work is supported by the SSF under the project WebSec (Ref. RIT17-0011).

Bibliography

Abadi, Martín et al. (1999). "A Core Calculus of Dependency". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, pp. 147–160. DOI: 10.1145/292540.292555. URL: https://doi.org/10.1145/292540.292555 (cit. on pp. 65, 67–69, 73, 76, 94, 97, 98).

Abel, Andreas and Jean-Philippe Bernardy (2020). "A unified view of modalities in type systems". In: *Proc. ACM Program. Lang.* 4.ICFP, 90:1–90:28. DOI: 10.1145/3408972. URL: https://doi.org/10.1145/3408972 (cit. on p. 99).

Abramsky, Samson and Radha Jagadeesan (2009). "Game Semantics for Access Control". In: *Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics, MFPS 2009, Oxford, UK, April 3-7, 2009*. Ed. by Samson Abramsky et al. Vol. 249. Electronic Notes in Theoretical Computer Science. Elsevier, pp. 135–156. DOI: 10.1016/j.entcs.2009.07.088. URL: https://doi.org/10.1016/j.entcs.2009.07.088 (cit. on pp. 65, 68, 98).

Abramsky, Samson, Pasquale Malacaria, et al. (1994). "Full Abstraction for PCF". In: *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*. Ed. by Masami Hagiya and John C. Mitchell. Vol. 789. Lecture Notes in Computer Science. Springer, pp. 1–15. DOI: 10.1007/3-540-57887-0_87. URL: https://doi.org/10.1007/3-540-57887-0%5C_87 (cit. on p. 98).

Algehed, Maximilian and Jean-Philippe Bernardy (2019). "Simple noninterference from parametricity". In: *Proc. ACM Program. Lang.* 3.ICFP, 89:1–89:22. DOI: 10.1145/3341693. URL: https://doi.org/10.1145/3341693 (cit. on p. 98).

Askarov, Aslan, Sebastian Hunt, et al. (2008). "Termination-Insensitive Noninterference Leaks More Than Just a Bit". In: *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*. Ed. by Sushil Jajodia and Javier López. Vol. 5283. Lecture Notes in Computer Science. Springer, pp. 333–348. DOI: 10.1007/978-3-540-88313-5_22. URL: https://doi.org/10.1007/978-3-540-88313-5%5C_22 (cit. on pp. 66, 71, 95).

Askarov, Aslan and Andrei Sabelfeld (2009). "Catch me if you can: permissive yet secure error handling". In: *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*. Ed. by Stephen Chong and David A. Naumann. ACM,

pp. 45–57. DOI: 10.1145/1554339.1554346. URL: https://doi.org/10.1145/1554339.1554346 (cit. on p. 83).

Atkey, Robert (2009). "Parameterised notions of computation". In: *J. Funct. Program.* 19.3-4, pp. 335–376. DOI: 10.1017/S095679680900728X. URL: https://doi.org/10.1017/S095679680900728X (cit. on p. 101).

Beck, Jon (1969). "Distributive laws". In: *Sem. on Triples and Categorical Homology Theory (ETH, Zürich, 1966/67)*. Springer, Berlin, pp. 119–140 (cit. on p. 79).

Borceux, Francis (1994). *Handbook of categorical algebra. 2*. Vol. 51. Encyclopedia of Mathematics and its Applications. Categories and structures. Cambridge University Press, Cambridge, pp. xviii+443. ISBN: 0-521-44179-X (cit. on p. 74).

Bowman, William J. and Amal Ahmed (2015). "Noninterference for free". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, pp. 101–113. DOI: 10.1145/2784731.2784733. URL: https://doi.org/10.1145/2784731.2784733 (cit. on p. 98).

Capretta, Venanzio (2005). "General recursion via coinductive types". In: *Log. Methods Comput. Sci.* 1.2. DOI: 10.2168/LMCS-1(2:1)2005. URL: https://doi.org/10.2168/LMCS-1(2:1)2005 (cit. on p. 94).

Chapman, James et al. (2019). "Quotienting the delay monad by weak bisimilarity". In: *Math. Struct. Comput. Sci.* 29.1, pp. 67–92. DOI: 10.1017/S0960129517000184. URL: https://doi.org/10.1017/S0960129517000184 (cit. on p. 95).

Denning, Dorothy E. (1976). "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5, pp. 236–243. DOI: 10.1145/360051.360056. URL: https://doi.org/10.1145/360051.360056 (cit. on p. 69).

Gaboardi, Marco et al. (2016). "Combining effects and coeffects via grading". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by Jacques Garrigue et al. ACM, pp. 476–489. DOI: 10.1145/2951913.2951939. URL: https://doi.org/10.1145/2951913.2951939 (cit. on p. 99).

Goguen, Joseph A. and José Meseguer (1982). "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, pp. 11–20. DOI: 10.1109/SP.1982.10014. URL: https://doi.org/10.1109/SP.1982.10014 (cit. on p. 65).

Gregersen, Simon Oddershede et al. (2021). "Mechanized logical relations for termination-insensitive noninterference". In: *Proc. ACM Program. Lang.*

5.POPL, pp. 1–29. DOI: 10.1145/3434291. URL: https://doi.org/10.1145/3434291 (cit. on p. 99).

Heintze, Nevin and Jon G. Riecke (1998). "The SLam Calculus: Programming with Secrecy and Integrity". In: *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. Ed. by David B. MacQueen and Luca Cardelli. ACM, pp. 365–377. DOI: 10.1145/268946.268976. URL: https://doi.org/10.1145/268946.268976 (cit. on p. 99).

Hirsch, Andrew K. and Ethan Cecchetti (2021). "Giving semantics to program-counter labels via secure effects". In: *Proc. ACM Program. Lang.* 5.POPL, pp. 1–29. DOI: 10.1145/3434316. URL: https://doi.org/10.1145/3434316 (cit. on p. 100).

Hunt, Sebastian et al. (2023). "Reconciling Shannon and Scott with a Lattice of Computable Information". In: *Proc. ACM Program. Lang.* 7.POPL, pp. 1987–2016. DOI: 10.1145/3571740. URL: https://doi.org/10.1145/3571740 (cit. on pp. 67, 68, 94, 97, 101).

Jacobs, Bart P. F. (2001). *Categorical Logic and Type Theory*. Vol. 141. Studies in logic and the foundations of mathematics. North-Holland. ISBN: 978-0-444-50853-9. URL: http://www.elsevierdirect.com/product.jsp?isbn=9780444508539 (cit. on p. 72).

Kammar, Ohad and Dylan McDermott (2018). "Factorisation Systems for Logical Relations and Monadic Lifting in Type-and-effect System Semantics". In: *Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2018, Dalhousie University, Halifax, Canada, June 6-9, 2018*. Ed. by Sam Staton. Vol. 341. Electronic Notes in Theoretical Computer Science. Elsevier, pp. 239–260. DOI: 10.1016/j.entcs.2018.11.012. URL: https://doi.org/10.1016/j.entcs.2018.11.012 (cit. on p. 83).

Katsumata, Shin-ya (2014). "Parametric effect monads and semantics of effect systems". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, pp. 633–646. DOI: 10.1145/2535838.2535846. URL: https://doi.org/10.1145/2535838.2535846 (cit. on pp. 76, 101).

Kavvos, G. A. (2019). "Modalities, cohesion, and information flow". In: *Proc. ACM Program. Lang.* 3.POPL, 20:1–20:29. DOI: 10.1145/3290333. URL: https://doi.org/10.1145/3290333 (cit. on pp. 65, 67–69, 71–74, 76, 84, 86, 97).

Lago, Ugo Dal and Francesco Gavazzo (2022). "A relational theory of effects and coeffects". In: *Proc. ACM Program. Lang.* 6.POPL, pp. 1–28. DOI: 10.1145/3498692. URL: https://doi.org/10.1145/3498692 (cit. on p. 99).

Lago, Ugo Dal, Francesco Gavazzo, and Paul Blain Levy (2017). "Effectful applicative bisimilarity: Monads, relators, and Howe's method". In: *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, pp. 1–12. DOI: 10.1109/LICS.2017.8005117. URL: https://doi.org/10.1109/LICS.2017.8005117 (cit. on p. 99).

Landauer, Jaisook and Timothy Redmond (1993). "A Lattice of Information". In: *6th IEEE Computer Security Foundations Workshop - CSFW'93, Franconia, New Hampshire, USA, June 15-17, 1993, Proceedings*. IEEE Computer Society, pp. 65–70. DOI: 10.1109/CSFW.1993.246638. URL: https://doi.org/10.1109/CSFW.1993.246638 (cit. on p. 98).

Levy, Paul Blain (2002). "Possible World Semantics for General Storage in Call-By-Value". In: *Computer Science Logic, 16th International Workshop, CSL 2002, 11th Annual Conference of the EACSL, Edinburgh, Scotland, UK, September 22-25, 2002, Proceedings*. Ed. by Julian C. Bradfield. Vol. 2471. Lecture Notes in Computer Science. Springer, pp. 232–246. DOI: 10.1007/3-540-45793-3_16. URL: https://doi.org/10.1007/3-540-45793-3%5C_16 (cit. on p. 101).

– (2004). *Call-By-Push-Value: A Functional/Imperative Synthesis*. Vol. 2. Semantics Structures in Computation. Springer. ISBN: 1-4020-1730-8 (cit. on p. 73).

Marmolejo, F., R. D. Rosebrugh, et al. (2002). "A basic distributive law". In: vol. 168. 2-3. Category theory 1999 (Coimbra), pp. 209–226. DOI: 10.1016/S0022-4049(01)00097-4. URL: https://doi.org/10.1016/S0022-4049(01)00097-4 (cit. on p. 78).

Marmolejo, F. and R. J. Wood (2010). "Monads as extension systems—no iteration is necessary". In: *Theory Appl. Categ.* 24 (cit. on p. 78).

Melliès, Paul-André and Noam Zeilberger (2015). "Functors are Type Refinement Systems". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, pp. 3–16. DOI: 10.1145/2676726.2676970. URL: https://doi.org/10.1145/2676726.2676970 (cit. on p. 72).

Møgelberg, Rasmus Ejlers and Alex Simpson (2007). "Relational Parametricity for Computational Effects". In: *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE

Computer Society, pp. 346–355. DOI: 10.1109/LICS.2007.40. URL: https://doi.org/10.1109/LICS.2007.40 (cit. on p. 98).

Moggi, Eugenio (1989). "Computational Lambda-Calculus and Monads". In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, pp. 14–23. DOI: 10.1109/LICS.1989.39155. URL: https://doi.org/10.1109/LICS.1989.39155 (cit. on pp. 66, 101).

– (1991). "Notions of Computation and Monads". In: *Inf. Comput.* 93.1, pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4. URL: https://doi.org/10.1016/0890-5401(91)90052-4 (cit. on pp. 73, 76, 80).

Pitts, Andrew M. and Ian David Bede Stark (1993). "Observable Properties of Higher Order Functions that Dynamically Create Local Names, or What's new?" In: *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings*. Ed. by Andrzej M. Borzyszkowski and Stefan Sokolowski. Vol. 711. Lecture Notes in Computer Science. Springer, pp. 122–141. DOI: 10.1007/3-540-57182-5_8. URL: https://doi.org/10.1007/3-540-57182-5%5C_8 (cit. on p. 101).

Plotkin, Gordon D. and John Power (2003). "Algebraic Operations and Generic Effects". In: *Appl. Categorical Struct.* 11.1, pp. 69–94. DOI: 10.1023/A:1023064908962. URL: https://doi.org/10.1023/A:1023064908962 (cit. on p. 81).

Pottier, François and Vincent Simonet (2003). "Information flow inference for ML". In: *ACM Trans. Program. Lang. Syst.* 25.1, pp. 117–158. DOI: 10.1145/596980.596983. URL: https://doi.org/10.1145/596980.596983 (cit. on p. 83).

Rajani, Vineet and Deepak Garg (2020). "On the expressiveness and semantics of information flow types". In: *J. Comput. Secur.* 28.1, pp. 129–156. DOI: 10.3233/JCS-191382. URL: https://doi.org/10.3233/JCS-191382 (cit. on p. 66).

Reynolds, John C. (1983). "Types, Abstraction and Parametric Polymorphism". In: *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. Ed. by R. E. A. Mason. North-Holland/IFIP, pp. 513–523 (cit. on pp. 98, 99).

Sabelfeld, Andrei and Andrew C. Myers (2003). "Language-based information-flow security". In: *IEEE J. Sel. Areas Commun.* 21.1, pp. 5–19. DOI: 10.1109/JSAC.2002.806121. URL: https://doi.org/10.1109/JSAC.2002.806121 (cit. on p. 87).

Sabelfeld, Andrei and David Sands (2001). "A Per Model of Secure Information Flow in Sequential Programs". In: *High. Order Symb. Comput.* 14.1, pp. 59–

91. DOI: 10.1023/A:1011553200337. URL: https://doi.org/10.1023/A:1011553200337 (cit. on pp. 89, 97).

Shikuma, Naokata and Atsushi Igarashi (2008). "Proving Noninterference by a Fully Complete Translation to the Simply Typed Lambda-Calculus". In: *Log. Methods Comput. Sci.* 4.3. DOI: 10.2168/LMCS-4(3:10)2008. URL: https://doi.org/10.2168/LMCS-4(3:10)2008 (cit. on p. 98).

Silver, Lucas et al. (2023). "Semantics for Noninterference with Interaction Trees". In: *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:29. DOI: 10.4230/LIPIcs.ECOOP.2023.29. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2023.29 (cit. on p. 90).

Sterling, Jonathan and Robert Harper (2022). "Sheaf Semantics of Termination-Insensitive Noninterference". In: *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel*. Ed. by Amy P. Felty. Vol. 228. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:19. DOI: 10.4230/LIPIcs.FSCD.2022.5. URL: https://doi.org/10.4230/LIPIcs.FSCD.2022.5 (cit. on pp. 65, 68, 98).

Tate, Ross (2013). "The sequential semantics of producer effect systems". In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, pp. 15–26. DOI: 10.1145/2429069.2429074. URL: https://doi.org/10.1145/2429069.2429074 (cit. on pp. 100, 101).

Tse, Stephen and Steve Zdancewic (2004). "Translating dependency into parametricity". In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*. Ed. by Chris Okasaki and Kathleen Fisher. ACM, pp. 115–125. DOI: 10.1145/1016850.1016868. URL: https://doi.org/10.1145/1016850.1016868 (cit. on p. 98).

Wadler, Philip and Peter Thiemann (2003). "The marriage of effects and monads". In: *ACM Trans. Comput. Log.* 4.1, pp. 1–32. DOI: 10.1145/601775.601776. URL: https://doi.org/10.1145/601775.601776 (cit. on p. 100).

Zdancewic, Stephan Arthur (2002). *Programming languages for information security*. Cornell University (cit. on p. 88).

C

Securing Asynchronous Exceptions

Carlos Tomé Cortiñas, Marco Vassena, and Alejandro Russo

33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020

Abstract Language-based information-flow control (IFC) techniques often rely on special purpose, ad hoc primitives to address different covert channels that originate in the runtime system, beyond the scope of language constructs. Since these piecemeal solutions may not compose securely, there is a need for a unified mechanism to control covert channels. As a *first step* towards this goal, we argue for the design of a general interface that allows programs to safely interact with the runtime system and the available computing resources. To coordinate the communication between programs and the runtime system, we propose the use of asynchronous exceptions (*interrupts*), which, to the best of our knowledge, have not been considered before in the context of IFC languages. Since asynchronous exceptions can be raised at any point during execution—often due to the occurrence of an external event—threads must temporarily *mask* them out when manipulating locks and shared data structures to avoid deadlocks and, therefore, breaking program invariants. Crucially, the naive combination of asynchronous exceptions with existing features of IFC languages (e.g. concurrency and synchronization variables) may open up new possibilities of information leakage. In this paper, we present MAC$^{\text{ASYNC}}$, a concurrent, statically enforced IFC language that, as a novelty, features asynchronous exceptions. We show how asynchronous exceptions easily enable (out of the box) useful programming patterns like speculative execution and some degree of resource management. We prove that programs in MAC$^{\text{ASYNC}}$ satisfy progress-sensitive noninterference and mechanize our formal claims in the AGDA proof assistant.

C.1. Introduction

Information-flow control (IFC) (Sabelfeld and Myers 2003) is a promising approach for preserving confidentiality of data. It tracks how data of different sensitivity levels (e.g. public or sensitive) flows within a program, and raises alarms when confidentiality might be at stake. This technology has been previously used to secure operating systems (e.g. Zeldovich et al. 2006; Vandebogart et al. 2007), web browsers (e.g. Stefan, Yang, et al. 2014; Yip et al. 2009), and several programming languages (e.g. Hedin, Birgisson, et al. 2014; Myers et al. 2006; Russo, Claessen, et al. 2008).

Most language-based approaches for IFC reason about constructions found in programs (e.g. variables, branches, and data structures), while often ignoring aspects of runtime systems which might create *covert channels* (e.g. (Buiras and Russo 2013; Mathias V. Pedersen and Askarov 2017; Vassena, Soeller, et al. 2019)) capable of producing leaks, e.g. through caches, parallelism, resource usage, etc. To deal with this problem, researchers have proposed security-aware runtime system designs (Vassena, Soeller, et al. 2019; Mathias Vorreiter Pedersen and Askarov 2019). However, building runtime systems is a major endeavour and these proposals have yet to be implemented. A more lightweight approach to securing runtime systems relies on *special-purpose language constructs* that coordinate the execution of programs with different components of the runtime—e.g. the garbage collector (Mathias V. Pedersen and Askarov 2017), the scheduler (Russo and Sabelfeld 2006), timeouts (Russo and Sabelfeld 2009), lazy evaluation (Vassena, Breitner, et al. 2017) and caches (Ferraiuolo et al. 2018).[1] While a step in the right direction, designing ad hoc constructs every time that some coordination with the runtime system is needed feels rather unsatisfactory—an observation that has also been made outside the security arena (Li, Marlow, et al. 2007; Sivaramakrishnan et al. 2016; Flatt and Findler 2004). In fact, implementing *hooks* in an existing runtime system requires specific knowledge of its internals and considerable expertise. Even worse, the composition of piecemeal security solutions may weaken or even break the security guarantees of the runtime system as a whole. These issues suggest the need for a unified mechanism to close covert channels in the runtime system. As a first step towards this goal, we believe that runtime systems should expose a general *IFC-aware interface* that allow IFC languages to systematically control and secure components of the runtime system. How should programs coordinate with the runtime system through this interface?

In the 70s, Unix-like operating systems conceived *signals* as a limited form

[1] In this last case, we are abusing the term runtime to denote "the rest of the system."

of inter process communication (IPC).[2] Signals are no more than asynchronous notifications sent to processes in order to notify them of the occurrence of events, where the origin of signals is either the kernel or other processes. Furthermore, when receiving a signal, process execution can be interrupted during any *nonatomic* instruction—and if the process has previously registered a signal handler, then that routine gets executed. If we think of the kernel as "the runtime" and of processes as our "programs", signals are exactly the mechanism needed to implement the interface that we need! In fact, and generally speaking, the idea of OS-signals have been already internalized by programming languages in the form of *asynchronous exceptions*.

Asynchronous exceptions are raised as a result of external events and can occur at *any point* of the program. As a result, they are considered so difficult to master that many languages (e.g. Python (Freund and M. P. Mitchell 2002) and Java (Oracle 2020)) either restrict or completely forbid programmers from using them. The main reason is that interrupting a program at any point might break, for instance, a datastructure invariant or result in holding a lock indefinitely—and it is not that clear how to get out of such situation.

Despite not being widely adopted in its full expressive power, asynchronous exceptions enable very useful programming patterns: *speculative execution* (i.e. a thread can spawn a child thread and later decide that it does not need the result and kill it), *timeouts*, and *resource management*.

Our Contributions

In this work, we present MACasync, a Haskell IFC library that extends the concurrent version of MAC (Russo 2015; Vassena, Russo, et al. 2018) with asynchronous exception. We formally prove progress-sensitive noninterference (PSNI) (Hedin and Sabelfeld 2012) for MACasync and provide mechanized proofs in Agda (Abel et al. 2005–) of all our claims as supplementary material to this work. We believe that the extension presented in this paper and its formal security guarantees extend to other Haskell IFC libraries (e.g. LIO (Stefan, Russo, J. C. Mitchell, et al. 2011)).

The semantics for asynchronous exceptions in MACasync is inspired by how asynchronous exception are modelled in Haskell (Marlow et al. 2001)—where a mechanism of *masking/unmasking* marks regions of code where asynchronous exceptions can be safely raised. However, allowing untrusted code to mask exceptions *arbitrarily* poses other security risks. For example, a rouge thread could abuse the masking mechanism to exhaust all available computing resources

[2] https://standards.ieee.org/content/ieee-standards/en/standard/1003_1-2017.html

and starve other threads in the system without the risk of being terminated. To avoid that, we propose a fine-grained (selective) masking/unmasking mechanism instead of the traditional *all-or-nothing* approaches, which disable all asynchronous exceptions inside handlers (Reppy 1990; Gabriel and McCarthy 1984). Furthermore, in contrast with Marlow et al. 2001, our design forbids raising multiple exceptions at the same time, which, we believe, can too easily disrupt programs in unpredictable ways. While an exception is raised, our language does not raise incoming exceptions, which are, instead, stored in a queue of pending exceptions and raised only when the current one has been handled.

From the security perspective, asynchronous exceptions follow the *no write-down* security check for IFC: when throwing an asynchronous exception, the security level of the source thread should flow to the security level of the recipient. The caveats are, however, in the formalization of masking/unmasking mechanisms and the noninterference proof. For example, it is important for security that asynchronous exception are *deterministically* inserted into the queue of pending exceptions. We utilize *term erasure* as the proof technique and leverage *double-step erasure* to deal with the complexity of our semantics (i.e. concurrency, synchronization variables and asynchronous exceptions), like in previous existing work (e.g. (Vassena, Russo, et al. 2018; Vassena, Buiras, et al. 2016; Vassena and Russo 2016)).

In summary, our list of contributions includes:

- An extension to MAC, called MACasync, to handle IFC-aware asynchronous exceptions in the presence of concurrency.
- Formal semantics, enforcement, and progress-insensitive noninterference guarantees for MACasync.
- Mechanized proofs of all our claims in approximately 3,000 lines of Agda.[3]
- We showcase MACasync and the new programming patterns enabled by asynchronous exceptions with two examples, in which we implement secure versions of (i) a speculative execution combinator, and (ii) a load-balancing controller for sensitive worker threads, respectively.

To the best of our knowledge, this work is the first account for asynchronous exceptions in concurrent IFC-systems. The rest of the paper is organized as follows. In Section C.2, we revisit MAC's API. Section C.3 presents MACasync by example. In Section C.4, we extend MAC's semantics to track asynchronous

[3] Available at `https://bitbucket.org/carlostome/mac-async`.

```
  -- Abstract types
data Labeled l τ
data MAC l τ
  -- Monadic structure for computations
instance Monad (MAC l)
  -- Core operations
label   :: l_L ⊑ l_H ⇒ τ → MAC l_L (Labeled l_H τ)
unlabel :: l_L ⊑ l_H ⇒ Labeled l_L τ → MAC l_H τ
```

Figure C.1.: Core API for MAC

exceptions. In Section C.5, we introduce asynchronous exceptions and the masking/unmasking mechanisms. Section C.6 presents our security guarantees. Section C.7 describes related work and Section C.8 concludes.

C.2. The MAC Information-Flow Control Library

To help readers get familiar with the MAC IFC library (Russo 2015), we give a brief overview of its API and programming model.

Security Lattice The information flow policies enforced by MAC are specified by a security lattice (Denning 1976), which defines a partial order between security levels (*labels*). These labels represent the sensitivity of program inputs and outputs and the *order* between them dictates which flows of information are allowed in a program. For example, the classic two-point lattice $L = (\{\mathtt{L}, \mathtt{H}\}, \sqsubseteq)$ classifies data as either *public* (L) or *secret* (H) and only prohibits sending secret inputs into public outputs, i.e. $\mathtt{H} \not\sqsubseteq \mathtt{L}$. In MAC, the security lattice is embedded in Haskell using standard features of the type system (Russo 2015). In particular, each security label is represented by an abstract datatype and valid flows of information (the $\sqsubseteq$ relation between labels) are encoded using typeclass constructs—see Figure C.1.

Security Types MAC enforces security statically by means of special types annotated with security labels. The abstract type *Labeled* l τ associates label l with data of type τ. For example, *pwd* :: *Labeled* H *String* is a secret string and *score* :: *Labeled* L *Int* is a public integer. The abstract type *MAC* l τ represents a side-effectful computation that manipulates data labelled with l and whose

result has type τ. MAC provides a monadic interface to help programmers write secure code. The basic primitives of the interface are *return* and *bind* (written as the infix operator $\gg=$). Primitive $return :: \tau \to MAC\ l\ \tau$ creates a computation that simply returns a value of type τ without causing side-effects. Primitive $(\gg=) :: MAC\ l\ \tau_1 \to (\tau_1 \to MAC\ l\ \tau_2) \to MAC\ l\ \tau_2$ chains two computations (at the same security level l) together, in a *sequence*. Specifically, program $m \gg= f$ takes the result obtained by executing m and *binds* it to function f, which produces the rest of the computation. Our examples use **do**-notation, Haskell syntactic sugar for monadic computations. For instance, we write **do** $x \leftarrow m$; $return\ (x+1)$ for the program $m \gg= \lambda x \to return\ (x+1)$, which increments by one the result returned by m.

Flows of Information In order to enforce information flow policies, MAC regulates the interaction between *MAC* computations and *Labeled* data. Computations cannot *write* and *read* labelled data directly, but must use special functions *label* and *unlabel* (Figure C.1). These functions create and read labelled data as long as these operations comply with specific security rules, known as *no write-down* and *no read-up* (Bell and LaPadula 1996). Intuitively, function *label* writes some data into a fresh, $l_{\mathtt{H}}$-labelled value as long as the decision to do so depends on less sensitive data, i.e. the computation is labelled with $l_{\mathtt{L}}$ such that $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$. (To help readers, we use subscripts in metavariables $l_{\mathtt{L}}$ and $l_{\mathtt{H}}$ to indicate that $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$). Dually, function *unlabel* allows $l_{\mathtt{H}}$-labelled computations to read data from lower security levels, i.e. data labelled with $l_{\mathtt{L}}$ such that $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$. In the type signatures of these functions, the precondition $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$ is a typeclass constraint, which must be statically satisfied when type checking programs. As a result, programs that attempt to leak secret data, e.g. via *implicit flows*, are ill-typed and rejected by the compiler. In particular, programs cannot branch on *secret* H-labelled data directly, but must use *unlabel* first to extract its content. Once unlabelled, secret data can only be manipulated within a computation labelled with H thanks to the type of *unlabel* and *bind*. Then, trying to use function *label* to create *public* L-labelled data triggers a type error that represents a violation of the no write-down rule. Specifically, an attempt to create public data from within a secret context generates an *unsatisfiable* type constraint $\mathtt{H} \sqsubseteq \mathtt{L}$, arising from the use of *label*.

MAC incorporates other kinds of resources (e.g. references and network sockets) in a similar way. Resources are encapsulated in *labelled* resources handlers and the API exposed to labelled computations is designed so that the read and write side-effects of each operation respect the *no read-up* and *no write-down* rules.

$$
\begin{aligned}
&\mathit{fork} :: l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}} \Rightarrow \mathit{MAC}\ l_{\mathtt{H}}\ () \rightarrow \mathit{MAC}\ l_{\mathtt{L}}\ ()\\
&\textbf{data}\ \mathit{MVar}\ l\ \tau\\
&\mathit{newMVar} :: l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}} \Rightarrow \tau \rightarrow \mathit{MAC}\ l_{\mathtt{L}}\ (\mathit{MVar}\ l_{\mathtt{H}}\ \tau)\\
&\mathit{putMVar}\ \ :: \mathit{MVar}\ l\ \tau \rightarrow \tau \rightarrow \mathit{MAC}\ l\ ()\\
&\mathit{takeMVar} :: \mathit{MVar}\ l\ \tau \rightarrow \mathit{MAC}\ l\ \tau
\end{aligned}
$$

Figure C.2.: Concurrent API for MAC

Concurrency Extending IFC languages with concurrency is a delicate task because threads provide attackers with new means to leak data. For example, the possibility of executing computations concurrently magnifies the bandwidth of the termination covert channel (Stefan, Russo, Buiras, et al. 2012). This channel enables brute force attacks in which threads try to guess the secret and enter into a loop to suppress public outputs if they succeed. Even worse, the combination of concurrency and shared resources can introduce subtle *internal-timing channels* (Smith and Volpano 1998). This covert channel is exploited by attacks that influence the (public) outcome of *data races* with secret data (Buiras and Russo 2013; Stefan, Russo, Buiras, et al. 2012). To support concurrency securely, MAC: (i) decouples computations that manipulate secret data from computations that can generate public outputs, and (ii) prevents threads (labelled computations) from affecting data races between threads at lower security levels. Primitive *fork* (Figure C.2) allows a $l_{\mathtt{L}}$-labelled computation to fork a thread at a higher security level, i.e. labelled with $l_{\mathtt{H}}$ such that $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$. Intuitively, forking constitutes a *write* operation and thus the type of *fork* enforces the no write-down rule. MAC does not implements threads directly, but relies on Haskell *green* (lightweight user-level) threads. These threads are managed by the GHC runtime system running a round-robin scheduler, which is compatible with the security guarantees of MAC (Vassena and Russo 2016; Vassena, Russo, et al. 2018).

Synchronization Variables MAC supports shared mutable state in the form of synchronization variables, following the style of Concurrent Haskell (Jones, Gordon, et al. 1996). The abstract type $\mathit{MVar}\ l\ \tau$ (Figure C.2) represents a synchronization variable that can be either *empty* or *full* with a value of type τ at security level l. Threads can create and atomically access synchronization variables with functions $\mathit{newMVar}$, $\mathit{putMVar}$ and $\mathit{takeMVar}$. Function $\mathit{newMVar}$ creates a synchronization variable initially full with the given value. (Like *label* and *fork*, function $\mathit{newMVar}$ performs a write side-effect, thus its

type signature has a similar security check). Functions *putMVar* and *takeMVar* allow threads to write and read shared variables *synchronously*. In particular, these functions *block* threads trying to read or write variables in the wrong "state". For example, function *putMVar* writes a value into an *empty* variable and blocks the thread if the variable is full. Dually, function *takeMVar* empties a *full* variable and returns its content and blocks the caller otherwise. Notice that *putMVar* and *takeMVar* perform both *read* and *write* side-effects: they must always read the variable to determine whether the caller should be blocked. Then, the *no read-up* and *no write-down* security rules imply that these functions are secure only when they operate within the same security level, i.e. both the variable and the computation are labelled with l (Russo 2015).

C.3. MACasync by Example

Asynchronous exceptions enable useful programming patterns that, to our knowledge, cannot be coded securely in any existing IFC language. We illustrate some of these idioms in MACASYNC, which extends MAC with three new primitives *throwTo*, *mask* (*unmask*), and *catch*. These primitives allow threads to (1) send signals to threads at higher security levels by throwing exceptions *asynchronously*, (2) *suppress* (*enable*) exceptions in specific regions of code, and (3) *react* to exceptions by running their corresponding exception handler, respectively.

Example C.3.1 (Speculative execution)**.** Imagine two implementations of the same algorithm whose performance depends on the input. Instead of settling for one, we could run both concurrently and just return the output of the first that finishes. At that point the thread computing the other algorithm may be killed since its result is no longer necessary. We can implement such a combinator for speculative execution in MACASYNC using asynchronous exceptions. First, we declare *Kill* :: *Exception* as a new exception, and define *kill t*, a function that sends exception *Kill* asynchronously to the thread identified by *t*.[4]

```
1 data Exception = Kill | ...
2 kill t = throwTo t Kill
```

Then, we define the combinator *speculate*, which receives two computations c_1 and c_2 to run speculatively.

[4] In MACASYNC, primitive *fork* returns the identifier of the child thread to the parent.

```
 3 speculate :: MAC l a → MAC l a → MAC l a
 4 speculate c1 c2 = do
 5   m ← newEmptyMVar
 6   t1 ← fork (c1 >>= putMVar m)
 7   t2 ← fork (c2 >>= putMVar m)
 8   r  ← takeMVar m
 9   kill t1; kill t2
10   return r
```

The combinator creates an *empty* synchronization variable m (line 5) and forks two threads (6–7), which run computations c_1 and c_2 concurrently and then write the result to m. When the combinator reads variable m (8), it *blocks* until either thread terminates and fills it with the result. When this happens, the combinator resumes, kills the children threads (one may still be running) (9), and returns the result (10).

Example C.3.2 (Thread pool)**.** This example presents the code of a *controller* thread that maintains a pool of *worker* threads to perform computations on a stream of incoming (sensitive) inputs. In this scheme, the controller thread manages the worker threads in the pool by reacting to asynchronous exceptions sent by other (public and secret) threads in the system. For example, when some secret input becomes available, a thread can send an exception $Input_{\mathtt{H}}$ *secret* to the controller thread, which extracts the secret data and forwards it to the first available worker thread to process it. Similarly, when the thread pool is no longer needed, it can be deallocated by sending the exception *Kill* to the controller, which then kills each worker thread in the pool. In the same way, the controller could be programmed to react to specific exceptions and carry out even more tasks (e.g. dynamically resizing the thread pool).

To set up this scheme, a thread calls function *initTP* (Figure C.3) to initialize the thread pool and start the controller thread. Function *initTP n f* allocates an empty synchronization variable m (line 6), forks a pool of n worker threads executing function f (line 7), collects their identifiers *ts*, and passes it to the controller thread (line 8). As new input becomes available, the controller writes it to the shared variable m, which is then read by one of the workers and its content processed via function f (line 11). To avoid getting killed in the middle of a computation, worker threads *mask* exception *Kill* while processing data, thus ensuring that they always complete on-going computations without aborting prematurely. It may seem erroneous to mask also instruction *takeMVar*: can this cause a worker thread to *block indefinitely* waiting for new input? No, in Concurrent Haskell, and MACasync, operations that can block indefinitely

```
1  type Data = ...
2  data Exception = Kill | Input_l (Labeled l Data) | ...
3  type Size = Int
4  initTP :: Size → (Data → MAC H ()) → MAC L (TId H)
5  initTP n f = do
6    m ← newEmptyMVar
7    ts ← forM [1 .. n] (λ_ → fork (worker f m))
8    fork (controller ts m)
9  worker :: (Data → MAC H ()) → MVar H Data → MAC H ()
10 worker f m = do
11   mask [Kill] (takeMVar m ≫= f)
12   worker f m
13 controller :: [TId H] → MVar H Data → MAC H ()
14 controller ts m =
15   let wait = newEmptyMVar ≫= takeMVar in
16   catch wait
17     [(Input_H secret,
18      mask [Input_H, Kill]
19           (do s ← unlabel secret
20               putMVar m s
21               unmask [Input_H, Kill] (controller ts m)))
22     ,(Kill,
23      mask [Input_H, Kill] (forM ts kill)) ]
```

Figure C.3.: Thread pool example

(like *takeMVar*) are *interruptible*, i.e. they can receive and raise asynchronous exceptions even in masked blocks (Marlow et al. 2001).

Function *controller ts m* implements a controller thread for the thread pool *ts* sharing variable *m*. As long as it receives no exception, the controller thread simply waits on an always-empty synchronization variable via *wait* (line 15). When the thread receives an exception, it resumes and executes the corresponding code in the list of exception handlers. In particular, when new secret input becomes available ($Input_H$ *secret*), it opens the secret (line 19) and writes it to variable *m* (line 20), so that the worker threads can process it. Notice that if variable *m* is *full* at this point, then some previous input is still waiting to be processed (all workers threads are busy) and the controller just

waits on the variable. As soon as a worker thread completes, it empties the variable containing the pending input, and the controller resumes by writing the variable; then it continues to wait for further exceptions. To avoid dropping any input, the controller thread masks exceptions *Kill* and $Input_{\mathtt{H}}$ (line 18) while processing requests. For example, if exceptions were not properly masked in that block of code, the controller could receive an exception, e.g. *Kill*, which would terminate the thread while trying to feed the last input received to the workers. Once done, the controller unmasks the exceptions again (line 21) and continues to wait for new input. After receiving and *eventually* raising the exception *Kill*, the controller thread propagates it to all the workers in the pool (line 23) and then terminates. Also in this case, the controller thread masks the other exceptions, which could otherwise prematurely terminate the controller and leave some of the worker threads alive.

The example, however, has a catch! Primitive *putMVar* may also block the controller indefinitely like *takeMVar*, and thus may likewise be *interrupted* and raise an exception, even if that exception is masked. As a result, the controller thread could also be interrupted on line 20 and drop the current input. To fix the program, we introduce the combinator *retry killed m ss*, which repeatedly attempts to fill variable *m* with the inputs pending in list *ss* while handling other exceptions.

```
24 retry :: Bool → [TId H] → MVar H Data
25                          → [Data] → MAC H ()
26 retry killed ts m [] =
27   if killed then forM ts kill; exit else return ()
28 retry killed m (s : ss) =
29   catch (putMVar m s)
30         [(Input_H secret,
31           do s' ← unlabel secret
32              if killed
33                 then retry killed ts m (s : ss)
34                 else retry killed ts m ((s : ss) + [s'])
35         ,(Kill, retry True ts m (s : ss))]
```

If further inputs are received while executing *retry*, the function appends them to list *ss* to avoid dropping them, and thus ensuring that they will eventually be delivered to the workers. If the controller receives exception *Kill* while retrying, the Boolean flag *killed* is switched on and further inputs are discarded. In this case, when all the inputs received before *Kill* are dispatched, the controller kills the worker threads and terminates with *exit* (line 27)—the function *retry*

assumes exceptions *Input* H and *Kill* are masked so this operation will not be interrupted. In conclusion, to repair the code of *controller*, we simply replace *putMVar m s* (line 20) with *retry False ts m* [*s*].

Even though relatively simple, these examples cannot be coded in IFC languages without support for asynchronous communication like MAC. In these languages, synchronous primitives (e.g. *MVar*) must be restricted to operate within a single security level for security reasons, as explained in Section C.2 and Vassena, Russo, et al. (2018). For instance, if only synchronous communication was available, then the *controller* thread from our second example could not receive commands from public (L-labelled) threads.[5]

C.4. Formal Semantics

C.4.1. Core of MACasync

From a security perspective, the interaction between synchronization variables, asynchronous exceptions, and exception masking is a delicate matter. MACASYNC implements these primitives on top of those provided by Concurrent HASKELL, whose runtime is not designed with security in mind. For example, the fact that a thread may be able to resume another by sending an asynchronous exception (Marlow et al. 2001) (as explained in the second example above) may introduce subtle internal timing covert channels that weaken the security guarantees of MAC. To rule that out, we extend the small-step semantics of MAC from Vassena, Russo, et al. (2018) with asynchronous exceptions and perform a rigorous, comprehensive security analysis of the whole language.

The core of MACASYNC is the standard call-by-name λ-calculus with Boolean and unit type (Figure C.4). We specify the side-effect free semantics of the core λ-calculus (e.g. function abstraction, application) as a small-step reduction relation, $t_1 \rightsquigarrow t_2$, which denotes that term t_1 reduces in one step to t_2. These reduction rules are standard and we completely omit them in this presentation.

$$
\begin{array}{lll}
\text{Types:} & \tau ::= () \mid \mathit{Bool} \mid \tau_1 \to \tau_2 \\
\text{Values:} & v ::= () \mid \mathit{True} \mid \mathit{False} \mid \lambda x.t \\
\text{Terms:} & t ::= t_1\ t_2 \mid \textbf{if}\ t\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \mid v
\end{array}
$$

Figure C.4.: Core syntax

[5] In MAC, a public thread could technically communicate asynchronously with a secret thread by updating a secret, mutable reference. However, these labelled references would inevitably introduce serious data races and thus do not represent a viable alternative.

The defining feature of MACasync is the security monad *MAC*, which encapsulates computations that may produce side-effects. Figure C.5 specifies the syntax and part of the semantics for the side-effectful constructs of the language. The small-step relation $t_1 \longrightarrow t_2$ denotes a single *sequential* step that brings term t_1 of type $\mathit{MAC}\ l\ \tau$ to t_2.

Rule (Unlabel$_1$) reduces term $\mathit{unlabel}\ t_1$ to $\mathit{unlabel}\ t_2$ by evaluating the argument through a *pure* semantics step $t_1 \rightsquigarrow t_2$. When the argument is evaluated, rule (Unlabel$_2$) extracts the content of the labelled value and returns it in the security monad.

$$
\begin{array}{lll}
\text{Types:} & \tau ::= \cdots \mid \mathit{MAC}\ l\ \tau \mid \mathit{Labeled}\ l\ \tau & \\
\text{Values:} & v ::= \cdots \mid \mathit{Labeled}\ t \mid \mathit{return}\ t & \\
\text{Terms:} & t ::= \cdots \mid \mathit{label}\ t \mid \mathit{unlabel}\ t \mid t_1 \gg\!= t_2 &
\end{array}
$$

$$
\frac{t_1 \rightsquigarrow t_2}{\mathit{unlabel}\ t_1 \longrightarrow \mathit{unlabel}\ t_2}\ (\textsc{Unlabel}_1)
\qquad
\mathit{unlabel}\ (\mathit{Labeled}\ t) \longrightarrow \mathit{return}\ t\ (\textsc{Unlabel}_2)
$$

Figure C.5.: Syntax and semantics of MACasync (excerpts)

C.4.2. Synchronization Variables

Figure C.6 extends MACasync with synchronization variables. The store Σ is partitioned by label into separate memory segments S, each consisting of a list of memory cells c, which can be either *full* with a term ($(\!|\, t \,|\!)$) or *empty* ($\otimes$). A value $\mathit{MVar}_l\ n$ denotes a synchronization variable that refers to the n-th cell of the l-labelled memory segment in the store.[6]

In rule (New), primitive $\mathit{newMVar}_l\ t$ allocates a new memory cell containing term t in the l-labelled segment of the store, at *fresh* address $n = |\Sigma(l)|$, i.e. $\Sigma[(l, n) \mapsto (\!|\, t \,|\!)]$, and returns the corresponding synchronization variable $\mathit{MVar}_l\ n$. Term $\mathit{putMVar}_l\ t_1\ t_2$ writes term t_2 into the *empty* cell pointed by the synchronization variable t_1. To do that, rule (Put$_1$) starts evaluating the variable t_1 through a pure semantics step $t_1 \rightsquigarrow t_1'$. When the variable is fully evaluated, e.g. $\mathit{MVar}_l\ n$, rule (Put$_2$) takes over and writes the given term t in the cell identified by (l, n), i.e. $\Sigma[(l, n) \mapsto (\!|\, t \,|\!)]$. Notice that the term steps only if the cell in the store Σ is initially *empty*, i.e. $(l, n) \mapsto \otimes\ \in\ \Sigma$. If the cell is *full*,

[6] Some terms in the calculus carry a label annotation that is inferred from its type. For example, the label l in $\mathit{MVar}_l\ n$ comes from its type $\mathit{MVar}\ l\ \tau$.

$$
\begin{array}{ll}
\text{Store:} & \Sigma \in \mathit{Label} \to \mathit{Memory} \\
\text{Memory:} & S ::= [\,] \mid c : S \\
\text{Cell:} & c ::= \otimes \mid (\!| t |\!) \\
\text{Addresses:} & n \in \mathbb{N} \\
\text{Types:} & \tau ::= \cdots \mid \mathit{MVar}\ l\ \tau \\
\text{Values:} & v ::= \cdots \mid \mathit{MVar}_l\ n \\
\text{Terms:} & t ::= \cdots \mid \mathit{newMVar}_l\ t \mid \mathit{takeMVar}\ t \\
 & \qquad \mid \mathit{putMVar}\ t_1\ t_2
\end{array}
$$

(New)
$$\frac{n = |\Sigma(l)|}{\Sigma, \mathit{newMVar}_l\ t \longrightarrow \Sigma[(l,n) \mapsto (\!| t |\!)], \mathit{return}\ (\mathit{MVar}_l\ n)}$$

(Put$_1$)
$$\frac{t_1 \rightsquigarrow t_1'}{\Sigma, \mathit{putMVar}\ t_1\ t_2 \longrightarrow \Sigma, \mathit{putMVar}\ t_1'\ t_2}$$

(Put$_2$)
$$\frac{(l,n) \mapsto \otimes \in \Sigma \qquad \Sigma' = \Sigma[(l,n) \mapsto (\!| t |\!)]}{\Sigma, \mathit{putMVar}\ (\mathit{MVar}_l\ n)\ t \longrightarrow \Sigma', \mathit{return}\ ()}$$

Figure C.6.: Syntax and semantics for synchronization variables

the term cannot be reduced by any other rule of the semantics and gets *stuck*, capturing the intended *blocking* behaviour of synchronization variables. We omit the rules for *takeMVar*, which follow a similar pattern (Vassena, Russo, et al. 2018).

C.4.3. Concurrency

Unlike previous concurrent incarnations of MAC, threads in MAC$_{\text{ASYNC}}$ can communicate with each other by sending signals in the form of asynchronous exceptions. To enable this form of communication, the runtime system assigns a unique thread identifier to each thread of the system. Thread identifiers are *opaque* to avoid leaking secret data through the number of threads in the system, and *labelled* to prevent sensitive threads from sending exceptions to threads at lower security levels. MAC$_{\text{ASYNC}}$ incorporates thread identifiers with values $\mathit{TId}_l\ n$ of the new primitive type $\mathit{TId}\ l$, whose label l represents

the static security level of the thread identified by n. Thread identifiers are also *unforgeable* and only generated automatically by the runtime system each time a new thread is forked.

$$\mathit{fork} :: l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}} \Rightarrow \mathit{MAC}\ l_{\mathtt{H}}\ () \rightarrow \mathit{MAC}\ l_{\mathtt{L}}\ (\mathit{TId}\ l_{\mathtt{H}})$$

Figure C.7 extends the sequential calculus of MACasync with concurrency primitives. To simplify our security analysis, term $\mathit{fork}_l\ t$ is annotated with the security label l of thread t of type $\mathit{MAC}\ l\ ()$. Similarly to Vassena, Russo, et al. (2018), we decorate the sequential reduction relation from above with *events*, which inform the top-level scheduler of the execution of sequential commands that have global effects. For example, event $\mathbf{fork}_l(t)$ indicates that thread t at security level l has been forked and event **step** denotes an uninteresting (*silent*) sequential step. Later, we extend the category of events to keep track of asynchronous exceptions as well. Sequential steps are also parameterized by a *thread id map* ϕ, which represents a source of fresh thread identifiers for each security level. The use of this map is exemplified by rule (Fork). Whenever a new thread is forked, e.g. $\mathit{fork}_l\ t$, we use the label annotation l to generate a fresh identifier $n = \phi(l)$, which is then returned in the monad wrapped in the constructor of thread identifiers, i.e. $\mathit{TId}_l\ n$.

Events:	$e ::= \mathbf{step} \mid \mathbf{fork}_l(t)$
Thread Id:	$n \in \mathbb{N}$
Thread Id Map	$\phi \in \mathit{Label} \rightarrow \mathit{Thread\ Id}$
Types:	$\tau ::= \cdots \mid \mathit{TId}\ l$
Values:	$v ::= \cdots \mid \mathit{TId}_l\ n$
Terms:	$t ::= \cdots \mid \mathit{fork}_l\ t$

(Fork)

$$\frac{n = \phi(l)}{\Sigma, \mathit{fork}_l\ t \xrightarrow{\mathbf{fork}_l(t)}_{\phi} \Sigma, \mathit{return}\ (\mathit{TId}_l\ n)}$$

Figure C.7.: Syntax and semantics of *fork*

Figure C.8 introduces the top-level semantics relation that formalizes how concurrent configurations evolve. Concurrent configurations are pairs $\langle \Sigma, \Theta \rangle$ consisting of the concurrent store Σ and a map of thread pools Θ. The thread pool map Θ maps each label of the lattice to the list of threads T_s at that security level, currently in the system. Each rule of the concurrent semantics

Configuration: $C ::= \langle \Sigma, \Theta \rangle$
Thread Pool Map: $\Theta \in \mathit{Label} \to \mathit{Thread\ Pool}$
Thread Pool: $T_s ::= [\,] \mid (th, T_s)$
Thread State: $th ::= t$

(SEQ)
$$\frac{\phi = \mathsf{nextId}(\Theta) \qquad \Sigma_1, t_1 \xrightarrow{\mathbf{step}}_{\phi} \Sigma_2, t_2}{l, n \vdash \langle \Sigma_1, \Theta[(l,n) \mapsto t_1] \rangle \hookrightarrow \langle \Sigma_2, \Theta[(l,n) \mapsto t_2] \rangle}$$

(FORK)
$$\frac{\phi = \mathsf{nextId}(\Theta_1) \qquad n' = \phi(l') \qquad \Theta_2 = \Theta_1[(l,n) \mapsto t_2] \qquad \Sigma, t_1 \xrightarrow{\mathbf{fork}_{l'}(t)}_{\phi} \Sigma, t_2}{l, n \vdash \langle \Sigma, \Theta_1[(l,n) \mapsto t_1] \rangle \hookrightarrow \langle \Sigma, \Theta_2[(l',n') \mapsto t] \rangle}$$

Figure C.8.: Syntax and semantics of concurrent MACASYNC

constructs the source of fresh thread identifiers ϕ from the thread pool map Θ of the initial configuration by means of the function $\mathsf{nextId}(\Theta) = \lambda l.|\Theta(l)|$.

A concurrent step $l, n \vdash c_1 \hookrightarrow c_2$ indicates that configuration c_1 steps to c_2, while running the thread identified (l, n), i.e. the n-th thread of the l-labelled thread pool. The particular scheduler used to determine which thread runs at every step is not very relevant for our discussion, therefore we omit it in our semantics. It suffices to say that the security guarantees of MACASYNC carry over for a wide range of deterministic schedulers (Vassena, Russo, et al. 2018) (as witnessed by our mechanized proofs) and include the Round Robin scheduler adopted in Concurrent HASKELL. The concurrent rules rely on sequential events to determine which step to take. For example, rule (SEQ) extracts the running thread from the thread pool, i.e. $\Theta[(l,n) \mapsto t_1]$, which steps *silently*, i.e. generating event **step**, and thus the rule only reinserts the thread term in the thread pool, i.e. $\Theta[(l,n) \mapsto t_2]$. In contrast, event $\mathbf{fork}_{l'}(t)$ in rule (FORK) indicates that the running thread has forked, therefore the rule reinserts the parent thread in the pool, i.e. $\Theta_2 = \Theta_1[(l,n) \mapsto t_2]$, and also adds its child at the corresponding security level l' and fresh index $n' = \phi(l')$, i.e. $\Theta_2[(l', n') \mapsto t]$.

C.5. Asynchronous Exceptions

MACASYNC supports sending and handling asynchronous exceptions by means of two new primitives *throwTo* and *catch*, see Figure C.9. Primitive *throwTo* $t\ \xi$ raises the exception ξ of abstract type χ *asynchronously* in the thread with identifier t. Intuitively, this operation constitutes a write effect, therefore MACASYNC restricts its API according to the *no write-down* rule to enforce security. To this end, the API ensures that the security label of the receiver thread ($l_{\mathtt{H}}$) is at least as sensitive as the label of the sender ($l_{\mathtt{L}}$) through the type constraint $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$. Once delivered and raised, asynchronous exceptions behave like synchronous exceptions. They disrupt the execution of the receiving thread in the usual way, with the exception bubbling up in the code of the thread and, if uncaught, eventually crashing it. Threads can recover from exceptions by wrapping regions of code in a *catch* block. The same mechanism, allows threads to *react* to asynchronous signals by handling exceptions appropriately. Primitive *catch* t *hs* takes as a parameter a computation t and a list containing pairs of exceptions and handlers. Then, if an exception ξ is raised during the execution of t, the handler corresponding to the first exception *matching* ξ in the list *hs*, if there is one, gets executed.

$$
\begin{aligned}
&\mathit{throwTo} :: l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}} \Rightarrow \mathit{TId}\ l_{\mathtt{H}} \to \chi \to \mathit{MAC}\ l_{\mathtt{L}}\ () \\
&\mathit{catch} \quad\ :: \mathit{MAC}\ l\ \tau \to [(\chi, \mathit{MAC}\ l\ \tau)] \to \mathit{MAC}\ l\ \tau
\end{aligned}
$$

Figure C.9.: MACASYNC API for asynchronous exceptions

Figure C.10 extends the calculus with value *raise* ξ, which indicates that the computation is in an *exceptional state*, and a new event $\mathbf{throw}_l(\xi, n)$, which instructs the runtime to deliver exception ξ to the thread identified by (l, n). To model how asynchronous exceptions propagate precisely, we add new rules both to the sequential and concurrent semantics. Rule (THROWTO$_1$) evaluates the thread identifier in term *throwTo* $t_1\ \xi$, which reduces to *throwTo* $t_2\ \xi$ through the pure step $t_1 \rightsquigarrow t_2$. (For simplicity, our model assumes that exceptions are already evaluated in terms, thus the rules do not need to reduce them). When the thread identifier is fully evaluated, i.e. it is of the form $\mathit{TId}_l\ n$, rule (THROWTO$_2$) generates event $\mathbf{throw}_l(\xi, n)$ and returns unit. The rule reflects the *nonblocking* behaviour of *throwTo*, which always succeeds as soon as the thread identifier is evaluated and regardless of the state of the receiving thread. This design decision has important security implications that we discuss further in Section C.5.3. Rule (CATCH$_1$) executes the computation t_1 in term *catch* t_1 *hs*.

If during the execution of t_1 the computation receives some exception ξ, and the exception propagates up to the exception handler, then the term reduces to *catch* (*raise* ξ) *hs* and rules ($\textsc{Catch}_{\xi 1}$) and ($\textsc{Catch}_{\xi 2}$) determine whether the exception gets handled or not. In these rules, function $\mathsf{first}(hs, \xi)$ searches for a handler corresponding to exception ξ in the list *hs*. To do so, the function traverses the list of exception-handler pairs *hs* left-to-right until it finds a pair whose left component is equal to exception ξ. If a handler for that exception is in the list, i.e. $h = \mathsf{first}(hs, \xi)$, then ($\textsc{Catch}_{\xi 1}$) passes control to it. If no handler matches the exception, i.e. $\emptyset = \mathsf{first}(hs, \xi)$, then rule ($\textsc{Catch}_{\xi 2}$) simply propagates the exception.

$$
\begin{array}{lll}
\text{Exceptions:} & \xi & \\
\text{Events:} & e & ::= \cdots \mid \mathbf{throw}_l(\xi, n) \\
\text{Values:} & v & ::= \cdots \mid \mathit{raise}\ \xi \\
\text{Handlers:} & \mathit{hs} & ::= [\,] \mid (\xi, t) : \mathit{hs} \\
\text{Terms:} & t & ::= \cdots \mid \mathit{throwTo}\ t\ \xi \mid \mathit{catch}\ t\ \mathit{hs}
\end{array}
$$

$$
(\textsc{ThrowTo}_1)\quad \frac{t_1 \rightsquigarrow t_2}{\Sigma, \mathit{throwTo}\ t_1\ \xi \xrightarrow{\mathbf{step}}_\phi \Sigma, \mathit{throwTo}\ t_2\ \xi}
$$

$$
(\textsc{ThrowTo}_2)\quad \frac{}{\Sigma, \mathit{throwTo}\ (\mathit{TId}_l\ n)\ \xi \xrightarrow{\mathbf{throw}_l(\xi,n)}_\phi \Sigma, \mathit{return}\ ()}
$$

$$
(\textsc{Catch}_1)\quad \frac{\Sigma_1, t_1 \xrightarrow{e}_\phi \Sigma_2, t_2}{\Sigma_1, \mathit{catch}\ t_1\ \mathit{hs} \xrightarrow{e}_\phi \Sigma_2, \mathit{catch}\ t_2\ \mathit{hs}}
\qquad
(\textsc{Catch}_{\xi 1})\quad \frac{h = \mathsf{first}(\mathit{hs}, \xi)}{\Sigma, \mathit{catch}\ (\mathit{raise}\ \xi)\ \mathit{hs} \xrightarrow{\mathbf{step}}_\phi \Sigma, h}
$$

$$
(\textsc{Catch}_{\xi 2})\quad \frac{\emptyset = \mathsf{first}(\mathit{hs}, \xi)}{\Sigma, \mathit{catch}\ (\mathit{raise}\ \xi)\ \mathit{hs} \xrightarrow{\mathbf{step}}_\phi \Sigma, \mathit{raise}\ \xi}
$$

Figure C.10.: Syntax and semantics for asynchronous exceptions

C.5.1. Masking Exceptions

Asynchronous exceptions are typically sent in response to external events such as user interrupts and exceeding resource limits. These exceptions can disrupt threads unpredictably, at any moment during their execution, and end up breaking code invariants and leaving shared data structures in an inconsistent state. For example, an incoming exception may crash a thread inside a critical section and cause it to hold a lock indefinitely, without the possibility of cleaning up. Therefore, writing robust code in the presence of asynchronous exceptions requires a mechanism to *temporarily* suppress exceptions in critical sections that must not be interrupted. Inspired by Marlow et al. (2001), MACASYNC sports two *scoped* combinators, *mask* and *unamsk*, to disable and enable specific exceptions in a given code region, respectively.[7]

$$\begin{aligned} &\mathit{mask} &&:: \chi \to \mathit{MAC}\ l\ \tau \to \mathit{MAC}\ l\ \tau \\ &\mathit{unmask} &&:: \chi \to \mathit{MAC}\ l\ \tau \to \mathit{MAC}\ l\ \tau \end{aligned}$$

Intuitively, primitive $\mathit{mask}\ \xi\ t$ runs computation t with exceptions ξ disabled. If such an exception is received during the execution of t, the exception is *not* dropped, but stored in a buffer of pending exceptions ξ_s and raised once the execution goes past the *mask* instruction. Term $\mathit{unmask}\ \xi\ t$ works the other way around and enables exceptions ξ while executing t. In general, whether an exception received by a thread should be raised immediately or temporarily suppressed depends on the *masking context* of the thread. Intuitively, the masking context at each execution point depends on the (nested) *mask* and *unmask* instructions crossed up to that point. For instance, if program $\mathit{unmask}\ \xi\ (\mathit{mask}\ \xi'\ t)$ receives exception ξ while executing t, and if $\xi \not\equiv \xi'$ and t does not contain any $\mathit{mask}\ \xi$ instruction, then the exception gets raised, i.e. $\mathit{unmask}\ \xi\ (\mathit{mask}\ \xi'\ (\cdots\ \mathit{raise}\ \xi\ \cdots))$.

Figure C.11 presents the sequential semantics of *mask* and *unmask*. The masking context M is a map from exceptions to Booleans, representing a bit vector that indicates which exceptions can be raised in the reduction steps. To keep track of exceptions, the sequential relation carries the list of pending exceptions ξ_s, on the left, and the list of remaining exceptions ξ_s' on the right of the arrow. Further, the arrow is annotated with the masking context of the thread (M). Rules (MASK) and (UNMASK) modify the masking context accordingly via functions $\mathsf{mask}(M, \xi) = \lambda\xi'.\xi \equiv \xi' \ \vee \ M(\xi')$ and $\mathsf{unmask}(M, \xi)$ (analogous). In particular, the rules reduce term $\mathit{mask}\ \xi\ t$

[7] Even though these primitives take only a single exception as an argument, they are equivalent to the multi-exception variants used in Section C.3, i.e. $\mathit{mask}\ [\xi_1, \xi_2]\ t$ behaves exactly like $\mathit{mask}\ \xi_1\ (\mathit{mask}\ \xi_2\ t)$.

Mask: $M \in \chi \to \mathit{Bool}$
Terms: $t \ ::= \cdots \mid \mathit{mask}\ \xi\ t \mid \mathit{unmask}\ \xi\ t$
Exception list: $\xi_s ::= [\,] \mid (\xi : \xi_s)$

(Mask)
$$\frac{M_2 = \mathsf{mask}(M_1, \xi) \qquad \Sigma_1, t_1, \xi_s \xrightarrow{e}_{(\phi, M_2)} \Sigma_2, t_2\ \xi_s'}{\Sigma_1, \mathit{mask}\ \xi\ t_1, \xi_s \xrightarrow{e}_{(\phi, M_1)} \Sigma_2, \mathit{mask}\ \xi\ t_2, \xi_s'}$$

(Unmask)
$$\frac{M_2 = \mathsf{unmask}(M_1, \xi) \qquad \Sigma_1, t_1, \xi_s \xrightarrow{e}_{(\phi, M_2)} \Sigma_2, t_2, \xi_s'}{\Sigma_1, \mathit{unmask}\ \xi\ t_1, \xi_s \xrightarrow{e}_{(\phi, M_1)} \Sigma_2, \mathit{unmask}\ \xi\ t_2, \xi_s'}$$

(Mask_1)
$$\Sigma, \mathit{mask}\ \xi\ (\mathit{return}\ t), \xi_s \xrightarrow{\mathbf{step}}_{(\phi, M)} \Sigma, \mathit{return}\ t, \xi_s$$

(Mask_ξ)
$$\Sigma, \mathit{mask}\ \xi\ (\mathit{raise}\ \xi'), \xi_s \xrightarrow{\mathbf{step}}_{(\phi, M)} \Sigma, \mathit{raise}\ \xi', \xi_s$$

Figure C.11.: Syntax and semantics of *mask* (*unmask* is similar)

(respectively *unmask* ξ t) by executing term t with modified mask M_2 obtained from disabling (enabling) exception ξ in the current mask M_1, i.e. $M_2 = \mathsf{mask}(M_1, \xi)$ ($M_2 = \mathsf{unmask}(M_1, \xi)$). When a nested, *masked* computation has completed, either successfully (*return* t) or not (*raise* ξ'), rules (Mask_1) and (Mask_ξ) simply propagate the result.

The masking context M and the list of pending exceptions ξ_s determine whether any exception in the list should be raised or not. To reflect that, we need to adapt the semantics rules for most constructs of the calculus. Figure C.12 shows the modifications for the monadic bind ($\gg\!=$). (The rules for the other constructs are modified in a similar way, we refer the reader to the Agda mechanization for details).

First, we define function $\mathsf{unmasked}_{\xi_s'}(M, \xi_s)$, which extracts from the list of pending exceptions ξ_s the *first* exception ξ that is *unmasked* in M, i.e. such that $\neg\, M(\xi)$, The function walks down the list recursively and accumulates the exceptions ξ that are *masked* in M, i.e. such that $M(\xi)$, in the list ξ_s', which is then returned together with the rest of the list ξ_s, when an unmasked exception is found. If all the exceptions in the list are masked, the function

$$\frac{\mathsf{unmasked}_{[\,]}(M, \xi_s) = (\xi, \xi_s')}{\Sigma, \mathit{return}\ t_1 \gg\!= t_2, \xi_s \xrightarrow{\mathbf{step}}_{(\phi, M)} \Sigma, \mathit{raise}\ \xi, \xi_s'} \quad (\textsc{Bind-Raise})$$

$$\frac{\mathsf{unmasked}_{[\,]}(M, \xi_s) = \emptyset}{\Sigma, \mathit{return}\ t_1 \gg\!= t_2, \xi_s \xrightarrow{\mathbf{step}}_{(\phi, M)} \Sigma, t_2\ t_1, \xi_s} \quad (\textsc{Bind}_2)$$

$$\Sigma, \mathit{raise}\ \xi \gg\!= t, \xi_s \xrightarrow{\mathbf{step}}_{(\phi, M)} \Sigma, \mathit{raise}\ \xi, \xi_s \quad (\textsc{Bind}_\xi)$$

Figure C.12.: *Masking* semantics of bind ($\gg\!=$)

simply returns $\emptyset$.

$$\mathsf{unmasked}_{\xi_s'}(M, \xi_s) = \begin{cases} \emptyset & \text{if } \xi_s = [\,] \\ (\xi, \xi_s' + \xi_s'') & \text{if } \xi_s = \xi : \xi_s'' \text{ and } \neg\, M(\xi) \\ \mathsf{unmasked}_{(\xi_s' + [\xi])}(M, \xi_s'') & \text{if } \xi_s = \xi : \xi_s'' \text{ and } M(\xi) \end{cases}$$

In the *elimination* rules of the semantics, we apply function $\mathsf{unmasked}_{[\,]}(M, \xi_s)$ to determine whether a pending exception should be raised. For example, if all exceptions are masked, i.e. $\mathsf{unmasked}_{[\,]}(M, \xi_s) = \emptyset$, then rule ($\textsc{Bind}_2$) steps as usual. In contrast, if an unmasked exception is pending, i.e. $\mathsf{unmasked}_{[\,]}(M, \xi_s) = (\xi, \xi_s')$, rule (Bind-Raise) raises it, i.e. *raise* ξ, and the thread steps with buffer ξ_s' where exception ξ has been removed.

Once raised, exceptions propagate *unconditionally* via rule ($\textsc{Bind}_\xi$), i.e. no further exceptions are raised until the current one is handled.

C.5.2. Concurrency and Synchronization Variables

The modifications needed for supporting asynchronous exceptions in the concurrent semantics are minimal. Figure C.14 extends the thread state *th* with the list of pending exceptions ξ_s and the initial masking context M. When a thread forks, the child thread *inherits* the masking context of the parent thread and runs with an initially empty list of exceptions. New rule (Throw) processes event $\mathbf{throw}_{l'}(\xi, n')$ by delivering exception ξ to the thread (t, ξ_s', M')

$$\frac{(l,n) \mapsto (\!|\, t \,|\!) \;\in\; \Sigma \qquad \mathsf{unmasked}_{[]}(M, \xi_s) = \emptyset \qquad \Sigma' = \Sigma[(l,n) \mapsto \otimes]}{\Sigma, \mathit{takeMVar}\ (\mathit{MVar}_l\ n), \xi_s \xrightarrow{\mathbf{step}}_{(\phi,M)} \Sigma', \mathit{return}\ t, \xi_s} \quad (\textsc{Take}_1)$$

$$\frac{(l,n) \mapsto c \;\in\; \Sigma \qquad \mathsf{unmasked}_{[]}(M, \xi_s) = (\xi, \xi_s')}{\Sigma, \mathit{takeMVar}\ (\mathit{MVar}_l\ n), \xi_s \xrightarrow{\mathbf{step}}_{(\phi,M)} \Sigma, \mathit{raise}\ \xi, \xi_s'} \quad (\textsc{Take-Raise-Unmasked})$$

$$\frac{(l,n) \mapsto \otimes \;\in\; \Sigma \qquad \mathsf{unmasked}_{[]}(M, \xi_s) = \emptyset \qquad \xi_s = \xi : \xi_s'}{\Sigma, \mathit{takeMVar}\ (\mathit{MVar}_l\ n), \xi_s \xrightarrow{\mathbf{step}}_{(\phi,M)} \Sigma, \mathit{raise}\ \xi, \xi_s'} \quad (\textsc{Take-Raise-Masked})$$

Figure C.13.: Synchronization variables and exceptions

Thread: $th ::= (t, \xi_s, M)$

$$\frac{\begin{array}{c}\phi = \mathsf{nextId}(\Theta_1) \qquad \Sigma, t_1, \xi_s \xrightarrow{\mathbf{throw}_{l'}(\xi,n')}_{(\phi,M)} \Sigma, t_2, \xi_s \\ \Theta_2 = \Theta_1[(l,n) \mapsto (t_2, \xi_s, M)] \qquad (l',n') \mapsto (t, \xi_s', M') \;\in\; \Theta_2\end{array}}{l, n \vdash \langle \Sigma, \Theta_1[(l,n) \mapsto (t_1, \xi_s, M)] \rangle \hookrightarrow \langle \Sigma, \Theta_2[(l',n') \mapsto (t, \xi_s' + [\xi], M')] \rangle} \quad (\textsc{Throw})$$

Figure C.14.: Extended concurrent semantics for asynchronous exceptions

identified by (l', n'). Since exceptions are processed in the same order as they are delivered, the rule inserts ξ at the end of the buffer ξ_s', i.e. $\xi_s' + [\xi]$.

Next, we introduce new rules that capture precisely the interaction between synchronization variables and asynchronous exceptions. As we explained before, Concurrent Haskell by design allows specific *blocking* operations to be *interrupted* by asynchronous exceptions, even if they are masked (Marlow et al. 2001). Therefore, our semantics resumes threads stuck on synchronization variables if *any* exception is pending. The rules in Figure C.13 formalize this requirement for primitive *takeMVar*, the rules for *putMVar* are symmetric. Rule ($\textsc{Take}_1$) covers the case where no unmasked exception is pending, i.e. $\mathsf{unmasked}_{[]}(M, \xi_s) = \emptyset$, and the thread can step because the variable is full, i.e. $(l,n) \mapsto (\!|\, t \,|\!) \;\in\; \Sigma$, and thus the rule returns its content t and empties the vari-

able, i.e. $\Sigma[(l,n) \mapsto \otimes]$. On the other hand, in rule (Take-Raise-Unmasked), an unmasked exception is pending, i.e. $\mathsf{unmasked}_{[]}(M, \xi_s) = (\xi, \xi_s')$, thus, regardless of the variable being full or empty, i.e. $(l,n) \mapsto c \in \Sigma$, the rule aborts the computation and raises the exception ξ without modifying the store. Lastly, in rule (Take-Raise-Masked), the variable is *empty*, i.e. $(l,n) \mapsto \otimes \in \Sigma$, and the thread should block and get stuck. However, an exception ξ is pending in the buffer ξ_s, i.e. $\xi_s = \xi : \xi_s'$, therefore—*regardless* of the masking context M—the thread is resumed by raising exception ξ. In the rule, the condition $\mathsf{unmasked}_{[]}(M, \xi_s) = \emptyset$ reveals that the pending exception that gets raised is *masked* and ensures that the semantics is *deterministic*. Without this premise, a thread with both unmasked and masked exceptions pending in its buffer could either step via rule (Take-Raise-Unmasked) and raise the first unmasked exception, or via rule (Take-Raise-Masked) and raise the first masked exception. The condition above removes the nondeterminism: if both masked and unmasked exceptions are pending, the first *unmasked* exception is raised via rule (Take-Raise-Unmasked).

C.5.3. Design Choices and Security

In this part we motivate some of the design choices that are key to the security guarantees of MAC$_{\text{ASYNC}}$ and that, we believe, can help programmers to write code that is more robust to asynchronous exceptions.

API of *throwTo* The type of *throwTo* (Figure C.9) restricts how threads are permitted to communicate asynchronously with each other to enforce security. Imagine an unrestricted version of *throwTo* called $\mathit{throwTo}^{\mathbf{leaky}}$, which—in clear violation of the *no write-down* security rule—allows secret threads to send exceptions to public threads. If MAC$_{\text{ASYNC}}$ exposed this leaky primitive, then an attacker could exploit it to leak secret data to a public thread through classic *implicit flows* attacks:

$$
\begin{array}{l}
\mathbf{do}\ \mathit{tid}_{\mathtt{L}} \leftarrow \mathit{fork}_{\mathtt{L}}\ (\mathbf{do}\ \mathit{catch}\ \mathit{loop}\ [(\xi, \mathit{print}_{\mathtt{L}}\ 1)]) \\
\quad \_ \quad \leftarrow \mathit{fork}_{\mathtt{H}}\ (\mathbf{do}\ s \leftarrow \mathit{unlabel}\ \mathit{secret} \\
\qquad\qquad\qquad\qquad \mathbf{if}\ s\ \mathbf{then}\ \mathit{throwTo}^{\mathbf{leaky}}\ \mathit{tid}_{\mathtt{L}}\ \xi \\
\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \ \mathit{return}\ ())
\end{array}
$$

The code above forks two threads, a public thread that waits for an asynchronous exception in a loop, and a secret thread that branches on secret data and sends an exception to the public thread in one branch. Since the secret thread sends an exception to the public thread only when the secret is true, the attacker

can easily learn its value by simply monitoring the public output of the public thread, which prints 1 only when an exception is raised. MACASYNC rejects such attacks by statically enforcing the no write-down rule in the API of *throwTo*, which would make the code above ill-typed.

Asynchronous *throwTo* In MACASYNC, primitive *throwTo* is itself an asynchronous operation. As rule (THROWTO$_2$) in Figure C.10 shows, primitive *throwTo* always returns immediately, without waiting for the receiver thread to raise the exception. This design choice follows a previous line of work on asynchronous exceptions for HASKELL Marlow et al. 2001, where the authors argue that the asynchronous semantics is easier to implement. Maybe surprisingly, the current implementation of Concurrent HASKELL with asynchronous exception in the GHC runtime provides only a *synchronous* version of *throwTo*.[8] Would MACASYNC be secure with a synchronous primitive $throwTo^{\mathbf{sync}}$? No, unfortunately the possibility of two threads synchronizing by throwing exceptions opens a new covert channel. Consider the following example, where $throwTo^{\mathbf{sync}}$ has synchronous semantics, i.e. $throwTo^{\mathbf{sync}}$ *blocks* the sender thread until the exception is raised in the receiver thread.

$$
\begin{array}{l}
\textbf{do } tid_{\mathtt{H}} \leftarrow fork_{\mathtt{H}}\ (\textbf{do } s \leftarrow unlabel\ secret \\
\qquad\qquad\qquad\qquad\quad \textbf{if } s \textbf{ then } mask\ \xi\ loop \textbf{ else } loop) \\
\quad no\_ops \\
\quad throwTo^{\mathbf{sync}}\ tid_{\mathtt{H}}\ \xi \\
\quad print_{\mathtt{L}}\ 0
\end{array}
$$

In the code above, the main public thread forks a secret thread, which branches on secret data and in one branch enters the masked block *mask* ξ *loop*. After waiting for a sufficient amount of time through *no_ops*, the public thread sends exception ξ *synchronously* to the secret thread. If the secret thread is looping in the masked block, the exception ξ will never be raised, causing the public thread to block forever on $throwTo^{\mathbf{sync}}$ and thus suppressing the final public output $print_{\mathtt{L}}$ 0. Then, the attacker can learn the value of the secret by simply observing (the lack of) the output 0 on the public channel.

As discussed in Section C.2 for *MVar*, synchronous communication primitives perform both read *and* write side-effects, therefore $throwTo^{\mathbf{sync}}$ cannot operate securely between threads at different security levels. Even though Concurrent HASKELL provides only the equivalent of $throwTo^{\mathbf{sync}}$, we can still derive a secure asynchronous implementation for *throwTo* by internally forking an

[8] `https://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Exception.html#v:throwTo`

isolated thread that calls the unsafe $throwTo^{\mathbf{sync}}$, i.e. we define $throwTo\ t\ \xi$ as $fork\ (throwTo^{\mathbf{sync}}\ t\ \xi\ \gg return\ ())$.

Reliable Exception Delivery In MACasync, threads store the received exceptions in the buffer where they remain until raised. Importantly, the exceptions are raised following the order in which they have been delivered, thus enabling threads to react to signals in the same order as they arise. Even though multiple exceptions can be pending in the buffer, our semantics ensures that new exceptions are not raised while the thread is in an exceptional state. This choice eliminates, by design, the risk of multiple simultaneous exceptions disrupting critical code in unpredictable ways. Once handled via the matching exception handler, the code resumes normal execution and any other pending exception may be raised. This ensures that all remaining exceptions, if not masked, will eventually be raised.

C.5.4. Relation to MAC

MACasync extends MAC with asynchronous exceptions (Vassena, Russo, et al. 2018). MAC features exception-handling primitives and classic exceptions, but these operate within individual threads and are intended to signal and recover from exceptional conditions arising only *internally*, due to the current state of the computation. Asynchronous exceptions are more expressive than regular exceptions. In addition to signaling (external) exceptional conditions, they enable a flexible signal-based communication mechanism. In MAC, threads can communicate with each other only synchronously and *indirectly*, through synchronization variables. Though possible, this communication mechanism is too cumbersome to use as it would require programmers to establish an appropriate communication protocol and change their code heavily, for example to ensure that all threads that need communicating share the same synchronization variable. Even worse, communication in this style is limited between threads at the same security level. In contrast, threads in MACasync can communicate *directly*, by sending exceptions to the identifier of the intended receiver thread, and also to threads at a different, more sensitive security level. MACasync leverages the mechanization of MAC in its security proofs. Modelling the semantics of asynchronous exceptions presented in this paper required substantial changes to the existing artifact. These changes include extending the syntax and semantics of the previous model with our new primitives, as well as carefully adapting the existing semantics rules to capture the semantics of *interruptible exceptions*.

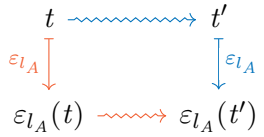


Figure C.15.: Single-step simulation

C.6. Security Guarantees

This section shows that MACASYNC satisfies progress-sensitive noninterference (PSNI). We begin by describing our proof technique based on *term erasure*. Then, we present two lemmas that are key to the progress-sensitive guarantees of the calculus and sketch the noninterference proof. We refer the interested reader to the AGDA mechanization for detailed proofs.

C.6.1. Term Erasure

Term erasure is a widely used technique to prove noninterference properties of information-flow control (IFC) languages (e.g. (Li and Zdancewic 2010; Stefan, Russo, J. C. Mitchell, et al. 2011; Stefan, Russo, Buiras, et al. 2012; Heule et al. 2015; Buiras, Vytiniotis, et al. 2015; Vassena and Russo 2016; Vassena, Russo, et al. 2018)). The technique takes its name from the *erasure function*, which removes secret data *syntactically* from program terms. To this end, the erasure function, written $\varepsilon_{l_A}(t)$, rewrites the subterms of t above the attacker's security level l_A to special term $\bullet$, which only reduces to itself. Once this function is defined, the technique relies on establishing a core property, a *simulation* between the execution of terms (and later configurations as well) and their erased counterpart. The simulation diagram in Figure C.15 illustrates this property for pure terms. The diagram shows that erasing the confidential parts of term t and then reducing the erased term $\varepsilon_{l_A}(t)$ along the orange path leads to the same term $\varepsilon_{l_A}(t')$ obtained along the blue path by first stepping from term t to term t' and then applying erasure, i.e. the diagram *commutes*. Intuitively, if term t leaked while stepping to t', then some data above security level l_A would remain in the erased term $\varepsilon_{l_A}(t')$, but it would be erased along the other path, in which t is first erased and then reduced, and thus the diagram would not commute.

C.6.2. Erasure Function

We define the erasure function for terms in Figure C.16. Since the sensitivity of many terms is determined by their type, the definition of the erasure function is type driven, i.e. we write $\varepsilon_{l_A}(t :: \tau)$ for the erasure of term t of type τ. (We omit the type of the term when it is irrelevant). Ground values are unaffected by the erasure function, e.g. $\varepsilon_{l_A}(()) = ()$, and most terms are erased homomorphically, e.g. $\varepsilon_{l_A}(t_1\ t_2) = \varepsilon_{l_A}(t_1)\ \varepsilon_{l_A}(t_2)$. The content of *secret* labelled values is removed, i.e. $\varepsilon_{l_A}(\mathit{Labeled}\ t{::}\mathit{Labeled}\ l\ t) = \mathit{Labeled}\ \bullet$ if $l\ \not\sqsubseteq\ l_A$, or erased homomorphically otherwise, i.e. $\varepsilon_{l_A}(\mathit{Labeled}\ t :: \mathit{Labeled}\ l\ t) = \mathit{Labeled}\ \varepsilon_{l_A}(t : \tau)$ if $l\ \sqsubseteq\ l_A$. Notice that terms of type $\mathit{MAC}\ l\ \tau$ (e.g. *mask*, *unmask*) are also erased homomorphically, despite the fact that the computation may be sensitive, i.e. even if $l\ \not\sqsubseteq\ l_A$. (The special erasure for primitive *throwTo* is explained below). Should not erasure rewrite these constructs to $\bullet$? Intuitively, these terms represent a *description* of a sensitive computation, which cannot leak data until it is inserted in a sequential configuration and executed. Since these terms can only execute when fetched from a thread pool, it is then sufficient to erase thread pools appropriately.

We define the erasure function for configurations, stores, thread pools, and thread states in Figure C.17. Configurations are erased component-wise, i.e. $\varepsilon_{l_A}(\langle \Sigma, \Theta \rangle) = \langle \varepsilon_{l_A}(\Sigma), \varepsilon_{l_A}(\Theta) \rangle$. Thread pools Θ containing secret threads are entirely removed by the erasure function, i.e. $\varepsilon_{l_A}(\Theta)(l) = \bullet$ if $l\ \not\sqsubseteq\ l_A$, while those containing thread pools are erased homomorphically, i.e. $\varepsilon_{l_A}(\Theta)(l) = \varepsilon_{l_A}(\Theta(l))$ if $l\ \sqsubseteq\ l_A$, where $\varepsilon_{l_A}([\,]) = [\,]$ and $\varepsilon_{l_A}(th, T_s) = (\varepsilon_{l_A}(th), \varepsilon_{l_A}(T_s))$. (The erasure function for memory stores and segments is similar). When some secret thread gets scheduled from an erased thread pool $\bullet$, a *dummy* thread $(\bullet, [\,], \lambda\_.\mathit{False})$ runs instead and simply loops. However, rewriting secret thread pools to $\bullet$ can disrupt operations involving thread identifiers. For example, an erased public thread using primitive *throwTo* to communicate with a secret thread gets *stuck*, since rule (Throw) would try to deliver an exception into thread pool $\bullet$. To recover from this situation, we apply the *two-step erasure* technique (Vassena and Russo 2016). This technique rewrites problematic terms, e.g. *throwTo*, to special, $\bullet$-annotated erased terms added to the calculus, i.e. $\mathit{throwTo}_\bullet$. The semantics of these *new* terms is then defined precisely to re-establish the core simulation property fundamental for security (Figure C.15). For example, term $\mathit{throwTo}_\bullet\ t\ \xi$ reduces just like *throwTo* in rules (ThrowTo$_1$) and (ThrowTo$_2$), thus respecting the simulation property of the sequential semantics. However, instead of generating a regular event $\mathbf{throw}_{l_H}(\xi, n)$, which would get the concurrent configuration stuck in rule (Throw), it generates a new event $\mathbf{throw}_{\bullet l_H}(\xi)$. Similarly, this event is handled by a new rule of the

$$
\begin{aligned}
&\varepsilon_{l_A}(()) = () \\
&\varepsilon_{l_A}(t_1\ t_2) = \varepsilon_{l_A}(t_1)\ \ \varepsilon_{l_A}(t_2) \\
&\varepsilon_{l_A}(\mathit{Labeled}\ t :: \mathit{Labeled}\ l\ t) \\
&\qquad = \begin{cases} \mathit{Labeled}\ \varepsilon_{l_A}(t) & \text{if } l_{\mathtt{H}} \sqsubseteq l_A \\ \mathit{Labeled}\ \bullet & \text{otherwise} \end{cases} \\
&\varepsilon_{l_A}(\mathit{mask}\ \xi\ t) = \mathit{mask}\ \xi\ \ \varepsilon_{l_A}(t) \\
&\varepsilon_{l_A}(\mathit{unmask}\ \xi\ t) = \mathit{unmask}\ \xi\ \ \varepsilon_{l_A}(t) \\
&\varepsilon_{l_A}(\mathit{throwTo}\ (t :: \mathit{TId}\ l_{\mathtt{H}})\ \xi :: \mathit{MAC}\ l_{\mathtt{L}}\ ()) \\
&\qquad = \begin{cases} \mathit{throwTo}\ \varepsilon_{l_A}(t)\ \ \xi & \text{if } l_{\mathtt{H}} \sqsubseteq l_A \\ \mathit{throwTo}_{\bullet}\ \varepsilon_{l_A}(t)\ \ \xi & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure C.16.: Erasure of terms (excerpts)

concurrent semantics, which simply drops the exception (no thread labelled $l_{\mathtt{H}} \not\sqsubseteq l_A$ can receive it), thus completing the simulation diagram of the concurrent step (THROW).

C.6.3. Progress-Sensitive Noninterference

The proof of progress-sensitive noninterference (PSNI) builds on two key properties of the concurrent relation: *deterministic reduction* and *erased simulation*.

Lemma C.6.1 (Deterministic Reduction)**.** *If* $c_1 \hookrightarrow c_2$ *and* $c_1 \hookrightarrow c_3$*, then* $c_2 \equiv c_3$*.*

The symbol $\equiv$ above denotes syntactic equality up to α-renaming, in our mechanized proofs we use De Bruijn indexes and syntactic equality. Determinism of the concurrent semantics is important for security, because it eliminates scheduler refinement attacks (Russo and Sabelfeld 2006).

The second lemma that we prove relates the reduction step of a thread in the concurrent semantics with the corresponding erased thread. If the security level l of the thread is below the level of the attacker, i.e. $l \sqsubseteq l_A$, then we construct a simulation diagram similar to that of Figure C.15, but for concurrent steps. Instead, if the security level of the thread is *not* observable by the attacker, i.e. $l \not\sqsubseteq l_A$, then the configurations before and after the step are *indistinguishable* to the attacker. This indistinguishability relation is called l_A-equivalence, written

$$\varepsilon_{l_A}(\langle \Sigma, \Theta \rangle) = \langle \varepsilon_{l_A}(\Sigma), \varepsilon_{l_A}(\Theta) \rangle$$
$$\varepsilon_{l_A}(\Sigma)(l) = \begin{cases} \varepsilon_{l_A}(S) & \text{if } l \sqsubseteq l_A \text{ and } S = \Sigma(l) \\ \bullet & \text{otherwise} \end{cases}$$
$$\varepsilon_{l_A}(\Theta)(l) = \begin{cases} \varepsilon_{l_A}(T_s) & \text{if } l \sqsubseteq l_A \text{ and } T_s = \Theta(l) \\ \bullet & \text{otherwise} \end{cases}$$
$$\varepsilon_{l_A}((t, \xi_s, M)) = (\varepsilon_{l_A}(t), \xi_s, M)$$

Figure C.17.: Erasure of configurations (excerpts)

$c_1 \approx_{l_A} c_2$, and defined as the kernel of the erasure function (Figure C.17), i.e. $\varepsilon_{l_A}(c_1) \equiv \varepsilon_{l_A}(c_2)$.

Lemma C.6.2 (Erased Simulation)**.** *Given a concurrent reduction step* $l, n \vdash c_1 \hookrightarrow c_1'$ *then*

- $l, n \vdash \varepsilon_{l_A}(c_1) \hookrightarrow \varepsilon_{l_A}(c_1')$*, if* $l \sqsubseteq l_A$*, or*
- $c_1 \approx_{l_A} c_1'$*, if* $l \not\sqsubseteq l_A$

Using Lemmas C.6.1 and C.6.2, we prove PSNI, where symbol $\hookrightarrow^*$ denotes the transitive reflexive closure of $\hookrightarrow$ as usual.

Theorem C.6.1 (Progress-Sensitive Noninterference)**.** *Given two well-typed concurrent configurations* c_1 *and* c_2*, such that* $c_1 \approx_{l_A} c_2$*, and a reduction step* $l, n \vdash c_1 \hookrightarrow c_1'$*, then there exists a configuration* c_2' *such that* $c_1' \approx_{l_A} c_2'$ *and* $c_2 \hookrightarrow^* c_2'$*.*

Proof. By cases on $l \sqsubseteq l_A$.

If $l \sqsubseteq l_A$ then in the configuration c_2 there is an l_A-equivalent thread identified by (l, n). Before that thread runs, however, there can be a *finite* number of high threads in c_2 scheduled before (l, n). After the high threads run, i.e. $c_2 \hookrightarrow^* c_2''$, for some configuration c_2'', the low thread is scheduled again, i.e. $l, n \vdash c_2'' \hookrightarrow c_2'$, for some other configuration c_2'. From Lemma C.6.2 (*erased simulation*) applied to the first set of steps, we obtain $c_2 \approx_{l_A} c_2''$ (all these steps involve threads above the attacker's level) and then $c_1 \approx_{l_A} c_2''$ follows by transitivity of the l_A-equivalence relation. Then, we apply Lemma C.6.2 again and conclude that $l, n \vdash \varepsilon_{l_A}(c_2'') \hookrightarrow \varepsilon_{l_A}(c_2')$, since $l \sqsubseteq l_A$ as well as $l, n \vdash \varepsilon_{l_A}(c_1) \hookrightarrow \varepsilon_{l_A}(c_1')$. By definition of l_A-equivalence, we derive $\varepsilon_{l_A}(c_1) \equiv \varepsilon_{l_A}(c_2'')$ from $c_1 \approx_{l_A} c_2''$ and

from Lemma C.6.1 (*deterministic reduction*) we conclude that $\varepsilon_{l_A}(c_1') \equiv \varepsilon_{l_A}(c_2')$, i.e. $c_1' \approx_{l_A} c_2'$.

If $l \not\sqsubseteq l_A$, then we apply Lemma C.6.2 and obtain $c_1 \approx_{l_A} c_1'$, thus $c_1' \approx_{l_A} c_2'$ for $c_2' = c_2$ by reflexivity and transitivity of l_A-equivalence. □

C.7. Related Work

Asynchronous Exceptions Mechanisms Asynchronous exceptions and signals allow developers to implement key functionalities of real world systems (e.g. speculative computation, timeouts, user interrupts, and enforcing resources bounds) robustly (Marlow et al. 2001; Marlow 2017). Surprisingly, support for asynchronous exceptions in concurrent programming languages differ considerably. For example, Java has deprecated fully asynchronous methods to stop, suspend, and resume threads because they can too easily break programs invariants without hope of recovery (Oracle 2020). Similarly, the interaction between synchronous exceptions and signals makes it hard to write robust signal handlers in Python (Freund and M. P. Mitchell 2002). Lacking robust asynchronous primitives, several programming languages and operating systems (e.g. Java, Modula3, and POSIX-compliant OS's) rely on *semi-asynchronous* communication as a workaround. With this approach, a thread sends a signal to another by setting special flags that must be polled periodically by the receiver. Even though programming in this model is less convenient, we believe that the principles proposed in this paper could be adapted for semi-asynchronous communication. The Standard ML of New Jersey (SML/NJ) features asynchronous signaling mechanisms based on first-class continuations (Reppy 1990). When a thread receives a signal, control is passed to the corresponding handler together with the interrupted state of the thread as a continuation. Then, the handler may decide to resume the execution of the interrupted thread or pass control to another thread. Erlang implements a special kind of asynchronous signaling. Threads can monitor each other through *bidirectional links*, which propagate the exit code of a thread to the other (Armstrong 2003). Multicore OCaml support asynchronous exceptions through algebraic effects and effects handlers (Dolan et al. 2017). Syme et al. (2011) extend F# with an asynchronous modality that changes the semantics of control-flow operators to use continuations, thus sparing programmers from writing asynchronous code in continuation-passing style. Bierman et al. (2012) port this approach to C# and additionally formalize it with a direct operational semantics and prove type-safety. Inoue et al. (2018) provide interruptible executions in Scala for context-aware (reactive) programming via an embedded domain specific

language based on workflows. Concurrent Haskell supports asynchronous exceptions with scoped (un)masking combinators (Marlow et al. 2001) and MACasync relies on them to provide secure API to untrusted code.

Semantics of Asynchronous Exceptions Jones, Reid, et al. (1999) present a semantics framework for reasoning about the correctness of compiler optimizations in the presence of (imprecise) exceptions for Haskell. Their framework can capture asynchronous exceptions as well, but it is based on denotational semantics and thus not suitable for reasoning about covert channels. Marlow et al. (2001) were the first to develop an operational semantics for asynchronous exceptions, which inspired ours and which we have extended to model fine-grained exception handlers. Their semantics is based on evaluation contexts (Felleisen and Hieb 1992), while ours is small-step to leverage the existing formalization and mechanization of MAC (Vassena, Russo, et al. 2018; Vassena and Russo 2016; Vassena, Breitner, et al. 2017). Hutton and Wright (2007) study an operational semantics for interrupts in the context of a basic terminating language without concurrency and I/O. Their goal is exploratory: they want to formally justify the source level semantics with respect to its compilation to a low-level language. Harrison et al. (2008) identify asynchronous exceptions as a computational effect and formalize them in a modular monadic model.

Covert Channels and Countermeasures Several runtime system features create subtle covert channels that weaken and sometimes completely break the security guarantees of IFC languages. When memory is shared between computations at different security levels, *garbage collection* cycles leak information via timing, even across network connections (Mathias V. Pedersen and Askarov 2017). To close this channel, memory should be partitioned by security level and each memory partition should be managed by an independent timing-sensitive garbage collector (see the garbage collector implemented in Zee for an example (Mathias Vorreiter Pedersen and Askarov 2019)). *Lazy evaluation* introduces a software level cache in the runtime system which creates an internal timing channel in concurrent Haskell IFC libraries (Buiras and Russo 2013). To close this channel, Vassena, Breitner, et al. (2017) design a runtime system primitive that *restricts sharing* between threads at different security levels. The same primitive can close the lazy covert channel in MACasync. General purpose runtime system automatically balance computing resources (CPU time, memory and cores) between running threads to achieve fairness, but, by doing so on multicore systems, they also internalize many external timing covert channels (Vassena, Soeller, et al. 2019). LIO_{PAR} is a runtime

system design that recovers security in multicore systems by making resource management hierarchical and explicit at the programming language level. Even though in LIO_{PAR} parent threads send asynchronous signals to kill children and reclaim computing resources, LIO_{PAR} does not support fine-grained exception handlers and masking primitives. Language-based predictive mitigation is a general technique to bound the leakage of timing channels (e.g. arising due to hardware caches) in programs (Zhang et al. 2012). Thibault and Askarov (2017) optimize this technique for a sequential programming language with asynchronous I/O, but their approach does not consider concurrency and asynchronous exceptions. Interruptible enclaves have been the target of several interrupt-based attacks (Bulck et al. 2017; He et al. 2018; Bulck et al. 2018) and Busi et al. (2020) propose *full abstraction* (Abadi 1999) as the desirable security criterion for extending processor with interruptible enclaves securely. Our security criterion (PSNIe) is simpler to prove and aligns with the expected security guarantees of MACasync. Intuitively, Busi et al. (2020) prove a more complex criterion because it ensures that the extended processor has no more vulnerabilities than the original, but that does not imply that neither processor satisfies some specific security property.

Secure Runtime Systems and Abstractions Systems that by design run untrusted programs (e.g. mobile code and plugins) must place adequate security mechanisms to impede buggy or malicious code from exhausting all available computing resources. KaffeOS is an extension of the Java runtime system that isolates *processes* and manage their computing resources (memory and CPU time) to prevent abuse (Back and Hsieh 2005). When a process exceeds its resource budget, KaffeOS kills it and reclaims its resources without affecting the integrity of the system. Cinder is an operating system for mobile devices that provides *reserves* and *taps* abstractions for storing and distributing energy (Roy et al. 2011). Using these abstractions, applications can delegate and subdivide their energy quota while maintaining energy isolation. Yang and Mazières (2014) extend GHC runtime systems with *resource containers*, an abstraction that enforce dynamic space limits according to an allocator-pays semantics. None of these systems enforce information flow policies except for Cinder, but we believe that secure API for asynchronous exceptions like those of MACasync could represent a basic building block to enforce them reliably.

Zee is an IFC language for implementing secure (timing-sensitive) runtime systems (Mathias Vorreiter Pedersen and Askarov 2019). The lack of asynchronous exceptions in Zee complicates the implementation of certain runtime system components, but we believe that Zee could support them by applying

the insights from this work. An interesting line of work aims at exposing safe high-level API to allow users to reprogram features of the runtime systems, e.g. concurrency primitives (Li, Marlow, et al. 2007), multicore schedulers (Sivaramakrishnan et al. 2016), and kill-safe abstractions (Flatt and Findler 2004). We believe that the primitives designed to remove covert channels in GHC and other runtime systems discussed above could be implemented following this approach.

C.8. Conclusions and Future Work

This work presents the first IFC language that support asynchronous exceptions securely. Embedded in Haskell, the IFC library MACasync provides primitives for fine-grained masking and unmasking of asynchronous exceptions, which enable useful programming patterns, that we showcased with two examples. We have formalized MACasync in 3,000 lines of Agda and proved progress-sensitive noninterference.

As future work, we plan to use MACasync to reason about the delivery of OS signals to threads. Specially, we will explore OS signals dedicated to alert about exhaustion of resources that cannot be easily partitioned (e.g. the battery in an IoT board). This scenario will demand the OS—which can be thought as just another thread—to send signals from higher to lower levels in the security lattice, thus opening up an information leakage channel which, we believe, needs to be mitigated.

Another direction for future work consists on using MACasync to build realistic systems. For instance, we expect MACasync to be able to provide an IFC-aware interface for GHC to control CPU usage by leveraging on previous work (Li, Marlow, et al. 2007; Sivaramakrishnan et al. 2016). Moreover, building realistic systems often involves mutually distrusts principals, where we expect *privileges* (Stefan, Russo, Mazières, et al. 2011; Waye et al. 2015) to restrict untrusted code from abusing our selective mask mechanism.

Acknowledgements

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011) and Octopi (Ref. RIT17-0023) as well as the Swedish research agency Vetenskapsrådet. This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity.

Bibliography

Abadi, Martín (1999). "Protection in Programming-Language Translations". In: *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*. Ed. by Jan Vitek and Christian Damsgaard Jensen. Vol. 1603. Lecture Notes in Computer Science. Springer, pp. 19–34. DOI: 10.1007/3-540-48749-2_2. URL: https://doi.org/10.1007/3-540-48749-2%5C_2 (cit. on p. 141).

[SW] Abel, Andreas et al., *Agda 2* 2005–. Chalmers University of Technology and Gothenburg University. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php (cit. on p. 112).

Armstrong, Joe (2003). "Making reliable distributed systems in the presence of hardware errors". PhD thesis. The Royal Institute of Technology Stockholm, Sweden (cit. on p. 139).

Back, Godmar and Wilson C. Hsieh (2005). "The KaffeOS Java runtime system". In: *ACM Trans. Program. Lang. Syst.* 27.4, pp. 583–630. DOI: 10.1145/1075382.1075383. URL: https://doi.org/10.1145/1075382.1075383 (cit. on p. 141).

Bell, David Elliott and Leonard J. LaPadula (1996). "Secure Computer Systems: A Mathematical Model, Volume II". In: *J. Comput. Secur.* 4.2/3, pp. 229–263 (cit. on p. 115).

Bierman, Gavin M. et al. (2012). "Pause 'n' Play: Formalizing Asynchronous C#". In: *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. Ed. by James Noble. Vol. 7313. Lecture Notes in Computer Science. Springer, pp. 233–257. DOI: 10.1007/978-3-642-31057-7_12. URL: https://doi.org/10.1007/978-3-642-31057-7%5C_12 (cit. on p. 139).

Buiras, Pablo and Alejandro Russo (2013). "Lazy Programs Leak Secrets". In: *Secure IT Systems - 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland, October 18-21, 2013, Proceedings*. Ed. by Hanne Riis Nielson and Dieter Gollmann. Vol. 8208. Lecture Notes in Computer Science. Springer, pp. 116–122. DOI: 10.1007/978-3-642-41488-6_8. URL: https://doi.org/10.1007/978-3-642-41488-6%5C_8 (cit. on pp. 111, 116, 140).

Buiras, Pablo, Dimitrios Vytiniotis, et al. (2015). "HLIO: mixing static and dynamic typing for information-flow control in Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, pp. 289–301. DOI: 10.1145/2784731.2784758. URL: https://doi.org/10.1145/2784731.2784758 (cit. on p. 135).

Bulck, Jo Van et al. (2017). "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. ACM, 4:1–4:6. DOI: 10.1145/3152701.3152706. URL: https://doi.org/10.1145/3152701.3152706 (cit. on p. 141).

– (2018). "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie et al. ACM, pp. 178–195. DOI: 10.1145/3243734.3243822. URL: https://doi.org/10.1145/3243734.3243822 (cit. on p. 141).

Busi, Matteo et al. (2020). "Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors". In: *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. IEEE, pp. 262–276. DOI: 10.1109/CSF49147.2020.00026. URL: https://doi.org/10.1109/CSF49147.2020.00026 (cit. on p. 141).

Denning, Dorothy E. (1976). "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5, pp. 236–243. DOI: 10.1145/360051.360056. URL: https://doi.org/10.1145/360051.360056 (cit. on p. 114).

Dolan, Stephen et al. (2017). "Concurrent System Programming with Effect Handlers". In: *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers*. Ed. by Meng Wang and Scott Owens. Vol. 10788. Lecture Notes in Computer Science. Springer, pp. 98–117. DOI: 10.1007/978-3-319-89719-6_6. URL: https://doi.org/10.1007/978-3-319-89719-6%5C_6 (cit. on p. 139).

Felleisen, Matthias and Robert Hieb (1992). "The Revised Report on the Syntactic Theories of Sequential Control and State". In: *Theor. Comput. Sci.* 103.2, pp. 235–271. DOI: 10.1016/0304-3975(92)90014-7. URL: https://doi.org/10.1016/0304-3975(92)90014-7 (cit. on p. 140).

Ferraiuolo, Andrew et al. (2018). "HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie et al. ACM, pp. 1583–1600. DOI: 10.1145/3243734.3243743. URL: https://doi.org/10.1145/3243734.3243743 (cit. on p. 111).

Flatt, Matthew and Robert Bruce Findler (2004). "Kill-safe synchronization abstractions". In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*. Ed. by William W. Pugh and Craig Chambers. ACM,

pp. 47–58. DOI: 10.1145/996841.996849. URL: https://doi.org/10.1145/996841.996849 (cit. on pp. 111, 142).

Freund, Stephen N. and Mark P. Mitchell (2002). *Safe Asynchronous Exceptions For Python*. Tech. rep. Williams College (cit. on pp. 112, 139).

Gabriel, Richard P. and John McCarthy (1984). "Queue-based Multi-processing Lisp". In: *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, Austin, Texas, USA, August 5-8, 1984*. Ed. by Robert S. Boyer et al. ACM, pp. 25–44. DOI: 10.1145/800055.802019. URL: https://doi.org/10.1145/800055.802019 (cit. on p. 113).

Harrison, William L. et al. (2008). "Asynchronous Exceptions as an Effect". In: *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings*. Ed. by Philippe Audebaud and Christine Paulin-Mohring. Vol. 5133. Lecture Notes in Computer Science. Springer, pp. 153–176. DOI: 10.1007/978-3-540-70594-9_10. URL: https://doi.org/10.1007/978-3-540-70594-9%5C_10 (cit. on p. 140).

He, Wenjian et al. (2018). "SGXlinger: A New Side-Channel Attack Vector Based on Interrupt Latency Against Enclave Execution". In: *36th IEEE International Conference on Computer Design, ICCD 2018, Orlando, FL, USA, October 7-10, 2018*. IEEE Computer Society, pp. 108–114. DOI: 10.1109/ICCD.2018.00025. URL: https://doi.org/10.1109/ICCD.2018.00025 (cit. on p. 141).

Hedin, Daniel, Arnar Birgisson, et al. (2014). "JSFlow: tracking information flow in JavaScript and its APIs". In: *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*. Ed. by Yookun Cho et al. ACM, pp. 1663–1671. DOI: 10.1145/2554850.2554909. URL: https://doi.org/10.1145/2554850.2554909 (cit. on p. 111).

Hedin, Daniel and Andrei Sabelfeld (2012). "A Perspective on Information-Flow Control". In: *Software Safety and Security - Tools for Analysis and Verification*. Ed. by Tobias Nipkow et al. Vol. 33. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, pp. 319–347. DOI: 10.3233/978-1-61499-028-4-319. URL: https://doi.org/10.3233/978-1-61499-028-4-319 (cit. on p. 112).

Heule, Stefan et al. (2015). "IFC Inside: Retrofitting Languages with Dynamic Information Flow Control (Extended Version)". In: *CoRR* abs/1501.04132. arXiv: 1501.04132. URL: http://arxiv.org/abs/1501.04132 (cit. on p. 135).

Hutton, Graham and Joel J. Wright (2007). "What is the meaning of these constant interruptions?" In: *J. Funct. Program.* 17.6, pp. 777–792. DOI:

10.1017/S0956796807006363. URL: https://doi.org/10.1017/S0956796807006363 (cit. on p. 140).

Inoue, Hiroaki et al. (2018). "ContextWorkflow: A Monadic DSL for Compensable and Interruptible Executions". In: *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*. Ed. by Todd D. Millstein. Vol. 109. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:33. DOI: 10.4230/LIPIcs.ECOOP.2018.2. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2018.2 (cit. on p. 139).

Jones, Simon L. Peyton, Andrew D. Gordon, et al. (1996). "Concurrent Haskell". In: *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. Ed. by Hans-Juergen Boehm and Guy L. Steele Jr. ACM Press, pp. 295–308. DOI: 10.1145/237721.237794. URL: https://doi.org/10.1145/237721.237794 (cit. on p. 116).

Jones, Simon L. Peyton, Alastair Reid, et al. (1999). "A Semantics for Imprecise Exceptions". In: *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*. Ed. by Barbara G. Ryder and Benjamin G. Zorn. ACM, pp. 25–36. DOI: 10.1145/301618.301637. URL: https://doi.org/10.1145/301618.301637 (cit. on p. 140).

Li, Peng, Simon Marlow, et al. (2007). "Lightweight concurrency primitives for GHC". In: *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*. Ed. by Gabriele Keller. ACM, pp. 107–118. DOI: 10.1145/1291201.1291217. URL: https://doi.org/10.1145/1291201.1291217 (cit. on pp. 111, 142).

Li, Peng and Steve Zdancewic (2010). "Arrows for secure information flow". In: *Theor. Comput. Sci.* 411.19, pp. 1974–1994. DOI: 10.1016/j.tcs.2010.01.025. URL: https://doi.org/10.1016/j.tcs.2010.01.025 (cit. on p. 135).

Marlow, Simon (Jan. 2017). *Asynchronous Exceptions in Practice*. URL: https://simonmar.github.io/posts/2017-01-24-asynchronous-exceptions.html (cit. on p. 139).

Marlow, Simon et al. (2001). "Asynchronous Exceptions in Haskell". In: *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*. Ed. by Michael Burke and Mary Lou Soffa. ACM, pp. 274–285. DOI: 10.1145/378795.378858. URL: https://doi.org/10.1145/378795.378858 (cit. on pp. 112, 113, 119, 121, 128, 131, 133, 139, 140).

[SW] Myers, Andrew C. et al., *Jif: Java information flow* 2006. URL: https://www.cs.cornell.edu/jif (cit. on p. 111).

Oracle (2020). *Java Thread Primitive Deprecation*. Oracle. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html (cit. on pp. 112, 139).

Pedersen, Mathias V. and Aslan Askarov (2017). "From Trash to Treasure: Timing-Sensitive Garbage Collection". In: *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, pp. 693–709. DOI: 10.1109/SP.2017.64. URL: https://doi.org/10.1109/SP.2017.64 (cit. on pp. 111, 140).

Pedersen, Mathias Vorreiter and Aslan Askarov (2019). "Static Enforcement of Security in Runtime Systems". In: *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, pp. 335–350. DOI: 10.1109/CSF.2019.00030. URL: https://doi.org/10.1109/CSF.2019.00030 (cit. on pp. 111, 140, 141).

Reppy, John H. (1990). *Asynchronous Signals is Standard ML*. Tech. rep. USA (cit. on pp. 113, 139).

Roy, Arjun et al. (2011). "Energy management in mobile devices with the cinder operating system". In: *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011*. Ed. by Christoph M. Kirsch and Gernot Heiser. ACM, pp. 139–152. DOI: 10.1145/1966445.1966459. URL: https://doi.org/10.1145/1966445.1966459 (cit. on p. 141).

Russo, Alejandro (2015). "Functional pearl: two can keep a secret, if one of them uses Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, pp. 280–288. DOI: 10.1145/2784731.2784756. URL: https://doi.org/10.1145/2784731.2784756 (cit. on pp. 112, 114, 117).

Russo, Alejandro, Koen Claessen, et al. (2008). "A library for light-weight information-flow security in haskell". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, pp. 13–24. DOI: 10.1145/1411286.1411289. URL: https://doi.org/10.1145/1411286.1411289 (cit. on p. 111).

Russo, Alejandro and Andrei Sabelfeld (2006). "Securing Interaction between Threads and the Scheduler". In: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. IEEE Computer Society, pp. 177–189. DOI: 10.1109/CSFW.2006.29. URL: https://doi.org/10.1109/CSFW.2006.29 (cit. on pp. 111, 137).

Russo, Alejandro and Andrei Sabelfeld (2009). "Securing Timeout Instructions in Web Applications". In: *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*. IEEE Computer Society, pp. 92–106. DOI: 10.1109/CSF.2009.16. URL: https://doi.org/10.1109/CSF.2009.16 (cit. on p. 111).

Sabelfeld, Andrei and Andrew C. Myers (2003). "Language-based information-flow security". In: *IEEE J. Sel. Areas Commun.* 21.1, pp. 5–19. DOI: 10.1109/JSAC.2002.806121. URL: https://doi.org/10.1109/JSAC.2002.806121 (cit. on p. 111).

Sivaramakrishnan, K. C. et al. (2016). "Composable scheduler activations for Haskell". In: *J. Funct. Program.* 26, e9. DOI: 10.1017/S0956796816000071. URL: https://doi.org/10.1017/S0956796816000071 (cit. on pp. 111, 142).

Smith, Geoffrey and Dennis M. Volpano (1998). "Secure Information Flow in a Multi-Threaded Imperative Language". In: *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. Ed. by David B. MacQueen and Luca Cardelli. ACM, pp. 355–364. DOI: 10.1145/268946.268975. URL: https://doi.org/10.1145/268946.268975 (cit. on p. 116).

Stefan, Deian, Alejandro Russo, Pablo Buiras, et al. (2012). "Addressing covert termination and timing channels in concurrent information flow systems". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*. Ed. by Peter Thiemann and Robby Bruce Findler. ACM, pp. 201–214. DOI: 10.1145/2364527.2364557. URL: https://doi.org/10.1145/2364527.2364557 (cit. on pp. 116, 135).

Stefan, Deian, Alejandro Russo, David Mazières, et al. (2011). "Disjunction Category Labels". In: *Information Security Technology for Applications - 16th Nordic Conference on Secure IT Systems, NordSec 2011, Tallinn, Estonia, October 26-28, 2011, Revised Selected Papers*. Ed. by Peeter Laud. Vol. 7161. Lecture Notes in Computer Science. Springer, pp. 223–239. DOI: 10.1007/978-3-642-29615-4_16. URL: https://doi.org/10.1007/978-3-642-29615-4%5C_16 (cit. on p. 142).

Stefan, Deian, Alejandro Russo, John C. Mitchell, et al. (2011). "Flexible dynamic information flow control in Haskell". In: *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. Ed. by Koen Claessen. ACM, pp. 95–106. DOI: 10.1145/2034675.2034688. URL: https://doi.org/10.1145/2034675.2034688 (cit. on pp. 112, 135).

Stefan, Deian, Edward Z. Yang, et al. (2014). "Protecting Users by Confining JavaScript with COWL". In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. Ed. by Jason Flinn and Hank Levy. USENIX Association, pp. 131–146. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/stefan (cit. on p. 111).

Syme, Don et al. (2011). "The F# Asynchronous Programming Model". In: *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*. Ed. by Ricardo Rocha and John Launchbury. Vol. 6539. Lecture Notes in Computer Science. Springer, pp. 175–189. DOI: 10.1007/978-3-642-18378-2_15. URL: https://doi.org/10.1007/978-3-642-18378-2%5C_15 (cit. on p. 139).

Thibault, Jérémy and Aslan Askarov (2017). *Language-based predictive mitigation for systems with asynchronous I/O*. Tech. rep. (cit. on p. 141).

Tomé Cortiñas, Carlos et al. (2020). "Securing Asynchronous Exceptions". In: *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. IEEE, pp. 214–229. DOI: 10.1109/CSF49147.2020.00023. URL: https://doi.org/10.1109/CSF49147.2020.00023 (cit. on p. 109).

Vandebogart, Steve et al. (2007). "Labels and event processes in the Asbestos operating system". In: *ACM Trans. Comput. Syst.* 25.4, p. 11. DOI: 10.1145/1314299.1314302. URL: https://doi.org/10.1145/1314299.1314302 (cit. on p. 111).

Vassena, Marco, Joachim Breitner, et al. (2017). "Securing Concurrent Lazy Programs Against Information Leakage". In: *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, pp. 37–52. DOI: 10.1109/CSF.2017.39. URL: https://doi.org/10.1109/CSF.2017.39 (cit. on pp. 111, 140).

Vassena, Marco, Pablo Buiras, et al. (2016). "Flexible Manipulation of Labeled Values for Information-Flow Control Libraries". In: *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*. Ed. by Ioannis G. Askoxylakis et al. Vol. 9878. Lecture Notes in Computer Science. Springer, pp. 538–557. DOI: 10.1007/978-3-319-45744-4_27. URL: https://doi.org/10.1007/978-3-319-45744-4%5C_27 (cit. on p. 113).

Vassena, Marco and Alejandro Russo (2016). "On Formalizing Information-Flow Control Libraries". In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*. Ed. by Toby C. Murray and Deian Stefan. ACM,

pp. 15–28. DOI: 10.1145/2993600.2993608. URL: https://doi.org/10.1145/2993600.2993608 (cit. on pp. 113, 116, 135, 136, 140).

Vassena, Marco, Alejandro Russo, et al. (2018). "MAC A verified static information-flow control library". In: *Journal of Logical and Algebraic Methods in Programming* 95, pp. 148–180. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2017.12.003. URL: https://www.sciencedirect.com/science/article/pii/S235222081730069X (cit. on pp. 112, 113, 116, 121, 123–125, 134, 135, 140).

Vassena, Marco, Gary Soeller, et al. (2019). "Foundations for Parallel Information Flow Control Runtime Systems". In: *Principles of Security and Trust - 8th International Conference, POST 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Ed. by Flemming Nielson and David Sands. Vol. 11426. Lecture Notes in Computer Science. Springer, pp. 1–28. DOI: 10.1007/978-3-030-17138-4_1. URL: https://doi.org/10.1007/978-3-030-17138-4%5C_1 (cit. on pp. 111, 140).

Waye, Lucas et al. (2015). "It's My Privilege: Controlling Downgrading in DC-Labels". In: *Security and Trust Management - 11th International Workshop, STM 2015, Vienna, Austria, September 21-22, 2015, Proceedings*. Ed. by Sara Foresti. Vol. 9331. Lecture Notes in Computer Science. Springer, pp. 203–219. DOI: 10.1007/978-3-319-24858-5_13. URL: https://doi.org/10.1007/978-3-319-24858-5%5C_13 (cit. on p. 142).

Yang, Edward Z. and David Mazières (2014). "Dynamic space limits for Haskell". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. Ed. by Michael F. P. O'Boyle and Keshav Pingali. ACM, pp. 588–598. DOI: 10.1145/2594291.2594341. URL: https://doi.org/10.1145/2594291.2594341 (cit. on p. 141).

Yip, Alexander et al. (2009). "Privacy-preserving browser-side scripting with BFlow". In: *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*. Ed. by Wolfgang Schröder-Preikschat et al. ACM, pp. 233–246. DOI: 10.1145/1519065.1519091. URL: https://doi.org/10.1145/1519065.1519091 (cit. on p. 111).

Zeldovich, Nickolai et al. (2006). "Making Information Flow Explicit in HiStar". In: *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. Ed. by Brian N. Bershad and Jeffrey C. Mogul. USENIX Association, pp. 263–278. URL: http://www.usenix.org/events/osdi06/tech/zeldovich.html (cit. on p. 111).

Zhang, Danfeng et al. (2012). "Language-based control and mitigation of timing channels". In: *ACM SIGPLAN Conference on Programming Language Design*

and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. Ed. by Jan Vitek et al. ACM, pp. 99–110. DOI: 10.1145/2254064.2254078. URL: https://doi.org/10.1145/2254064.2254078 (cit. on p. 141).

D

Simple Noninterference by Normalization

Carlos Tomé Cortiñas and Nachiappan Valliappan

Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, CCS 2019, London, United Kingdom, November 11-15, 2019

Abstract Information-flow control (IFC) languages ensure programs preserve the confidentiality of sensitive data. *Noninterference*, the desired security property of such languages, states that public outputs of programs must not depend on sensitive inputs. In this paper, we show that noninterference can be proved using normalization. Unlike arbitrary terms, normal forms of programs are well-principled and obey useful syntactic properties—hence enabling a simpler proof of noninterference. Since our proof is syntax-directed, it offers an appealing alternative to traditional semantic based techniques to prove noninterference.

In particular, we prove noninterference for a static IFC calculus, based on Haskell's SecLib library, using normalization. Our proof follows by straightforward induction on the structure of normal forms. We implement normalization using *normalization by evaluation* and prove that the generated normal forms preserve semantics. Our results have been verified in the Agda proof assistant.

D.1. Introduction

Information-flow control (IFC) is a security mechanism which guarantees confidentiality of sensitive data by controlling how information is allowed to flow in a program. The guarantee that programs secured by an IFC system do not leak sensitive data is often proved using a property called *noninterference*. Noninterference ensures that an observer authorized to view the output of a program (pessimistically called the attacker) cannot infer any sensitive data handled by it. For example, suppose that the type $\mathsf{Int}_{\mathtt{H}}$ denotes a secret integer and $\mathsf{Bool}_{\mathtt{L}}$ denotes a public Boolean. Now consider a program f with the following type:

$$f : \mathsf{Int}_{\mathtt{H}} \to \mathsf{Bool}_{\mathtt{L}}$$

For this program, noninterference ensures that f outputs the same Boolean for any given integer.

To prove noninterference, we must show that the public output of a program is not affected by varying the secret input. This has been achieved using many techniques including *term erasure* based on dynamic operational semantics (Li and Zdancewic 2010; Russo et al. 2008; Stefan et al. 2011; Vassena and Russo 2016), denotational semantics (Abadi et al. 1999; Kavvos 2019), and *parametricity* (Tse and Zdancewic 2004; Bowman and Ahmed 2015; Algehed 2018). In this paper, we show that noninterference can also be proved by normalizing programs using the static or *residualising* semantics (Lindley 2005) of the language.

If a program returns the same output for any given input, it must be the case that it does not depend on the input to compute the output. Thus proving noninterference for a program which receives a secret input and produces a public output, amounts to showing that the program behaves like a *constant* program. For example, proving noninterference for the program f consists of showing that it is equivalent to either $\lambda\, x\,.\, \mathit{true}$ or $\lambda\, x\,.\, \mathit{false}$; it is immediately apparent that these functions do not depend on the secret input x. But how can we prove this for *any* arbitrary definition of f?

The program f may have been defined as the simple function $\lambda\, x\,.\,(\mathit{not}\ \mathit{false})$ or perhaps the more complex function $\lambda\, x\,.\,((\lambda\, y\,.\,\mathit{snd}\ (x\ ,\ y))\ \mathit{true})$. Observe, however, that both these programs can be normalized to the equivalent function $\lambda\ x\ .\ \mathit{true}$. In general, although terms in the language may be arbitrarily complex, their *normal forms* (such as $\lambda\, x\,.\,\mathit{true}$) are not. They are simpler, thus well-suited for showing noninterference.

The key idea in this paper is to normalize terms, and prove noninterference by simple structural induction on their normal forms. To illustrate this, we prove

noninterference for a static IFC calculus, which we shall call λ_{SEC}, based on Haskell's SecLib library by Russo et al. We present the typing rules and static semantics for λ_{SEC} by extending Moggi's *computational metalanguage* (Moggi 1991) (Section D.2). We identify normal forms of λ_{SEC}, and establish syntactic properties about a normal form's dependency on its input (Section D.3). Using these properties, we show that the normal forms of program f are $\lambda\, x\,.\, true$ or $\lambda\, x\,.\, false$—as expected (Section D.4).

To prove noninterference for all terms using normal forms, we implement normalization for λ_{SEC} using *normalization by evaluation* (NbE) (Berger and Schwichtenberg 1991) and prove that it preserves the static semantics (Section D.5). Using normalization, we prove noninterference for program f and further generalize this proof to *all* terms in λ_{SEC} (Section D.6)—including, for example, a program which operates on both secret and public values such as $\mathsf{Bool}_{\mathsf{L}} \times \mathsf{Bool}_{\mathsf{H}} \to \mathsf{Bool}_{\mathsf{L}} \times \mathsf{Bool}_{\mathsf{H}}$. Finally, we conclude by discussing related work and future directions (Section D.7).

Unlike earlier proofs, our proof shows that noninterference is an inherent property of the normal forms of λ_{SEC}. Since the proof is primarily type and syntax-directed, it provides an appealing alternative to typical semantics based proof techniques. All the main theorems in this paper have been mechanized in the proof assistant Agda.[1]

D.2. The λ_{SEC} Calculus

In this section we present λ_{SEC}, a static IFC calculus that we shall use as the basis for our proof of noninterference. It models the pure and terminating fragment of the IFC library SecLib[2] for Haskell, and is an extension of the calculus developed by Russo et al. (2008) with sum types. SecLib is a lightweight implementation of static IFC which allows programmers to incorporate untrusted third-party code into their applications while ensuring that it does not leak sensitive data. Below, we recall the public interface (API) of SecLib:

$$
\begin{aligned}
&\mathsf{data}\ S\ (l :: \mathit{Lattice})\ a \\
&\mathit{return} :: a \to S\ l\ a \\
&(\gg\!=) \quad :: S\ l\ a \to (a \to S\ l\ b) \to S\ l\ b \\
&\mathit{up} \qquad :: l_{\mathsf{L}} \sqsubseteq l_{\mathsf{H}} \Rightarrow S\ l_{\mathsf{L}}\ a \to S\ l_{\mathsf{H}}\ a
\end{aligned}
$$

[1] https://github.com/carlostome/ni-nbe
[2] `https://hackage.haskell.org/package/seclib`

Similar to other static IFC libraries in Haskell such as LIO (Stefan et al. 2011) or MAC (Vassena, Russo, et al. 2018), SecLib's security guarantees rely on exposing the API to the programmer while hiding the underlying implementation. Programs written against the API and the *safe* parts of the language (Terei et al. 2012) are guaranteed to be *secure-by-construction*; the library enforces security statically through types. As an example, suppose that we have the two-point security lattice (see (Denning 1976)) $\{\mathtt{L}, \mathtt{H}\}$ where the only disallowed flow is from secret ($\mathtt{H}$) to public ($\mathtt{L}$), denoted $\mathtt{H} \not\sqsubseteq \mathtt{L}$. The following program written using the SecLib API is well-typed and—intuitively—secure:

$$
\begin{aligned}
&\mathit{example} :: S\ \mathtt{L}\ \mathsf{Bool} \rightarrow S\ \mathtt{H}\ \mathsf{Bool} \\
&\mathit{example}\ p\ =\ \mathit{up}\ (p \gg\!= \lambda\, b \rightarrow \mathit{return}\ (\mathit{not}\ b))
\end{aligned}
$$

The function *example* negates the Bool that it receives as input and upgrades its security level from public to secret. On the other hand, had the program tried to downgrade the secret input to public—clearly violating the policy of the security lattice—the typechecker would have rejected the program as ill-typed.

The Calculus λ_{SEC} is a simply typed λ-calculus (STLC) with a base (uninterpreted) type, unit type, product and sum types, and a security monad type for every security level in a set of labels (denoted by Label). The set of labels may be a lattice, but our development only requires it to be a preorder on the relation $\sqsubseteq$. Throughout the rest of this paper, we use the labels $l_{\mathtt{L}}$ and $l_{\mathtt{H}}$ and refer to them as *public* and *secret*, although they represent levels in an arbitrary security lattice such that $l_{\mathtt{H}} \not\sqsubseteq l_{\mathtt{L}}$. Figure D.1 defines the syntax of terms, types and contexts of λ_{SEC}.

In addition to the standard introduction and elimination constructs for unit, products and sums in STLC, λ_{SEC} uses the constructs return, let and up for the security monad $\mathsf{S}\ l\ \tau$, which mirrors S from SecLib. Note that our presentation favours let, as in Moggi (1989), over the Haskell bind ($\gg\!=$), although both presentations are equivalent—i.e. $t \gg\!= \lambda\, x\,.\, u$ can be encoded as $\mathsf{let}\ x = t\ \mathsf{in}\ u$.

The typing rules for return and let, shown in Figure D.2, ensure that computations over labelled values in the security monad $\mathsf{S}\ l\ \tau$ do not leak sensitive data. The construct return allows the programmer to tag a value of type τ with security label l; and bind enforces that sequences of computations over labelled values stay at the same security level.

Further, the calculus models the *up* combinator in SecLib as the construct up. Its purpose is to relabel computations to higher security levels. The rule Up, shown in Figure D.2, statically enforces that information can only flow from $l_{\mathtt{L}}$ to $l_{\mathtt{H}}$ in agreement with the security policy $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$. The rest of the

$$
\begin{array}{lll}
\text{Label } l\ ,\ l_{\mathtt{H}}\ ,\ l_{\mathtt{L}} & & \\
\text{Context } \Gamma\ \Delta\ \Sigma & ::= & \emptyset \mid \Gamma\ ,\ x : \tau \\
\text{Type } \tau\ \tau_1\ \tau_2 & ::= & \tau_1 \Rightarrow \tau_2 \mid \iota \mid () \\
& & \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \\
& & \mid \mathsf{S}\ l\ \tau \\
\text{Term } t\ s\ u & ::= & x \mid \lambda\, x\,.\,t \mid t\ s \mid () \\
& & \mid <t\ ,\ s> \mid \mathsf{fst}\ t \mid \mathsf{snd}\ t \\
& & \mid \mathsf{left}\ t \mid \mathsf{right}\ t \\
& & \mid \mathsf{case}\ t\ (\mathsf{left}\ x_1 \to s)\ (\mathsf{right}\ x_2 \to u) \\
& & \mid \mathsf{return}\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ u \mid \mathsf{up}\ t
\end{array}
$$

Figure D.1.: The λ_{SEC} calculus

$\boxed{\Gamma \vdash t : \tau}$

$$
\textsc{Return}\ \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{return}\ t : \mathsf{S}\ l\ \tau}
\qquad
\textsc{Up}\ \frac{\Gamma \vdash t : \mathsf{S}\ l_{\mathtt{L}}\ \tau \qquad l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}}{\Gamma \vdash \mathsf{up}\ t : \mathsf{S}\ l_{\mathtt{H}}\ \tau}
$$

$$
\textsc{Let}\ \frac{\Gamma \vdash t : \mathsf{S}\ l\ \tau_1 \qquad \Gamma\ ,\ x : \tau_1 \vdash s : \mathsf{S}\ l\ \tau_2}{\Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ s : \mathsf{S}\ l\ \tau_2}
$$

Figure D.2.: Type system of λ_{SEC} (excerpts)

typing rules for λ_{SEC} are standard (Pierce 2002), and thus omitted here. For a full account we refer the reader to our Agda formalization.

For completeness, the function *example* from earlier can be encoded in the λ_{SEC} calculus as follows:[3]

$$\mathit{example} \;=\; \lambda\, s\,.\,\mathsf{up}\ (\mathsf{let}\ b = s\ \mathsf{in}\ \mathsf{return}\ (\mathit{not}\ b))$$

Static Semantics The static semantics of λ_{SEC} is defined as a set of equations relating terms of the same type typed under the same environment. The equations characterize pairs of λ_{SEC} terms that are equivalent based on β-reduction,

[3] In λ_{SEC}, the type Bool is encoded as $() + ()$ with $\mathit{false} = \mathsf{left}\ ()$ and $\mathit{true} = \mathsf{right}\ ()$.

η-expansion and other monadic operations. We present the equations for return and let constructs of the monadic type S (à la Moggi (1991)) in Figure D.3, and further extend this with equations for the up primitive in Figure D.4. The remaining equations—including β and η rules for other types, and permutation rules for commuting case conversions—are fairly standard (Lindley 2005; Abel and Sattler 2019), and can be found in the Agda formalization. As customary, we use the notation $t_1\ [x/t_2]$ for capture-avoiding substitution of the term t_2 for variable x in term t_1.

$\boxed{\Gamma \vdash t_1 \approx t_2 : \tau}$

β-S
$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma\ ,\ x : \tau \vdash t_2 : \mathsf{S}\ l\ \tau}{\Gamma \vdash \mathsf{let}\ x = (\mathsf{return}\ t_1)\ \mathsf{in}\ t_2 \approx t_2\ [x/t_1] : \mathsf{S}\ l\ \tau}$$

η-S
$$\frac{\Gamma \vdash t : \mathsf{S}\ l\ \tau}{\Gamma \vdash t \approx \mathsf{let}\ x = t\ \mathsf{in}\ (\mathsf{return}\ x) : \mathsf{S}\ l\ \tau}$$

γ-S
$$\frac{\Gamma \vdash t_1 : \mathsf{S}\ l\ \tau_1 \qquad \Gamma\ ,\ x : \tau_1 \vdash t_2 : \mathsf{S}\ l\ \tau_2 \qquad \Gamma\ ,\ x : \tau_1\ ,\ y : \tau_2 \vdash t_3 : \mathsf{S}\ l\ \tau_3}{\Gamma \vdash \mathsf{let}\ x = (\mathsf{let}\ y = t_1\ \mathsf{in}\ t_2)\ \mathsf{in}\ t_3 \approx \mathsf{let}\ y = t_1\ \mathsf{in}\ (\mathsf{let}\ x = t_2\ \mathsf{in}\ t_3) : \mathsf{S}\ l\ \tau_3}$$

Figure D.3.: Static semantics of λ_{SEC} (return and let)

The up primitive induces equations regarding its interaction with itself and other constructs in the security monad. In Figure D.4, we make the auxiliary condition of up and the label of return explicit using subscripts for better clarity. These equations can be understood as follows:

- Rule δ_1-S. applying up over let is equivalent to distributing it over the subterms of let.
- Rule δ_2-S. applying up on an term labelled as return t is equivalent to relabelling t with the final label.
- Rule δ_{trans}-S. applying up twice is equivalent to applying it once using the transitivity of the relation $\sqsubseteq$.
- Rule δ_{refl}-S. applying up using the reflexive relation $l \sqsubseteq l$ is equivalent to not applying it.

$\boxed{\Gamma \vdash t_1 \approx t_2 : \tau}$

$$\frac{\Gamma \vdash t : \mathsf{S}\ l_{\mathtt{L}}\ \tau_1 \qquad \Gamma, x : \tau_1 \vdash u : \mathsf{S}\ l_{\mathtt{L}}\ \tau_2 \qquad p : l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}}{\Gamma \vdash \mathsf{up}_p\ (\mathsf{let}\ x = t\ \mathsf{in}\ u) \approx \mathsf{let}\ x = (\mathsf{up}_p\ t)\ \mathsf{in}\ (\mathsf{up}_p\ u) : \mathsf{S}\ l_{\mathtt{H}}\ \tau}\ \delta_1\text{-}\mathsf{S}$$

$$\frac{\Gamma \vdash t : \tau \qquad p : l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}}{\Gamma \vdash \mathsf{up}_p\ (\mathsf{return}_{\mathtt{L}}\ t) \approx \mathsf{return}_{\mathtt{H}}\ t : \mathsf{S}\ l_{\mathtt{H}}\ \tau}\ \delta_2\text{-}\mathsf{S}$$

$$\frac{\Gamma \vdash t : \mathsf{S}\ l_{\mathtt{L}}\ \tau \qquad p : l_{\mathtt{L}} \sqsubseteq l_{\mathtt{M}} \qquad q : l_{\mathtt{M}} \sqsubseteq l_{\mathtt{H}} \qquad r = \text{trans-}\sqsubseteq\ p\ q}{\Gamma \vdash \mathsf{up}_q\ (\mathsf{up}_p\ t) \approx \mathsf{up}_r\ t : \mathsf{S}\ l_{\mathtt{H}}\ \tau}\ \delta_{\text{TRANS}}\text{-}\mathsf{S}$$

$$\frac{\Gamma \vdash t : \mathsf{S}\ l\ \tau \qquad p : l \sqsubseteq l}{\Gamma \vdash \mathsf{up}_p\ t \approx t : \mathsf{S}\ l\ \tau}\ \delta_{\text{REFL}}\text{-}\mathsf{S}$$

Figure D.4.: Static semantics of λ_{SEC} (up)

D.3. Normal Forms of λ_{SEC}

As discussed in Section D.1, our proof of noninterference utilizes syntactic properties of normal forms, and hence relies on normalizing terms in the language. Normal forms are a restricted subset of terms in the λ_{SEC} calculus which intuitively corresponds to terms that cannot be normalized further. The syntax of normal forms is defined using two well-typed interdependent syntactic categories: *neutral* forms as $\Gamma \vdash_{\text{ne}} t : \tau$ (Figure D.5) and normal forms as $\Gamma \vdash_{\text{nf}} t : \tau$ (Figure D.6). Neutral forms are a special case of normal forms which depend entirely on the typing context (e.g. a variable).

Since the definition of neutral and normal forms are merely a syntactic restriction over terms, they can be embedded back into terms of λ_{SEC} using a *quotation* function $\ulcorner n \urcorner$. This embedding can be implemented for neutrals and normal forms by simply mapping them to their term counterparts.

Neutral Forms The neutral forms are terms which are characterized by a property called *neutrality*, which is stated as follows:

Property D.3.1 (Neutrality)**.** For a given neutral form of type $\Gamma \vdash_{\text{ne}} \tau$,

$\boxed{\Gamma \vdash_{\text{ne}} t : \tau}$

$$\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash_{\text{ne}} x : \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash_{\text{ne}} t : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash_{\text{nf}} s : \tau_1}{\Gamma \vdash_{\text{ne}} t\ s : \tau_2} \qquad \textsc{Fst}\ \frac{\Gamma \vdash_{\text{ne}} t : \tau_1 \times \tau_2}{\Gamma \vdash_{\text{ne}} \mathsf{fst}\ t : \tau_1}$$

$$\textsc{Snd}\ \frac{\Gamma \vdash_{\text{ne}} t : \tau_1 \times \tau_2}{\Gamma \vdash_{\text{ne}} \mathsf{snd}\ t : \tau_2}$$

Figure D.5.: Neutral forms

neutrality states that the type τ must occur as a *subformula* of a type in the context Γ.

For instance, given a neutral form $\Gamma \vdash_{\text{ne}} n : \mathsf{Bool}$, neutrality states that the type Bool must occur as a subformula of some type in the typing context Γ. An example of such a context is $\Gamma = [x : () \Rightarrow \mathsf{Bool}\ ,\ y : \mathsf{S}\ l_{\text{H}}\ \iota]$. The notion of a subformula, originally defined for logical propositional formulas in proof theory (Troelstra and Schwichtenberg 2000), can also be defined for types as follows:

Definition D.3.1 (Subformula)**.** For some types τ, τ_1 and τ_2; a subformula of a type is defined as:

- τ is a subformula of τ
- τ is a subformula of $\tau_1 \otimes \tau_2$ if τ is a subformula of τ_1 or τ is a subformula of τ_2, where $\otimes$ denotes the binary type operators $\times$, $+$ and $\Rightarrow$.

The type Bool occurs as a subformula in the typing context $[() \Rightarrow \mathsf{Bool}\ ,\ \mathsf{S}\ l_{\text{H}}\ \iota]$ since the type Bool is a subformula of the type $() \Rightarrow \mathsf{Bool}$. Note, however, that the type ι does not occur as a subformula in this context since ι is not a subformula of the type $\mathsf{S}\ l_{\text{H}}\ \iota$ by the above definition.

Normal Forms Intuitively, normal forms of type $\Gamma \vdash_{\text{nf}} \tau$ are characterized as terms of type $\Gamma \vdash \tau$ that cannot be *reduced* further using the static semantics. Precisely, a normal form is a term obtained by systematically applying the equations defined by the relation $\approx$ in a specific order to a given term. We leave the exact order of applying the equations unspecified since we only require that there *exists* a normal form for every term—we prove this later in

$\boxed{\Gamma \vdash_{\mathrm{nf}} t : \tau}$

$$\textsc{Unit}\ \frac{}{\Gamma \vdash_{\mathrm{nf}} () : ()} \qquad \textsc{Lam}\ \frac{\Gamma\,,\, x : \tau_1 \vdash_{\mathrm{nf}} t : \tau_2}{\Gamma \vdash_{\mathrm{nf}} \lambda\, x.\, t : \tau_1 \Rightarrow \tau_2} \qquad \textsc{Base}\ \frac{\Gamma \vdash_{\mathrm{ne}} t : \iota}{\Gamma \vdash_{\mathrm{nf}} t : \iota}$$

$$\textsc{Ret}\ \frac{\Gamma \vdash_{\mathrm{nf}} t : \tau}{\Gamma \vdash_{\mathrm{nf}} \mathsf{return}\ t : \mathsf{S}\ l\ \tau}$$

$$\textsc{LetUp}\ \frac{l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}} \qquad \Gamma \vdash_{\mathrm{ne}} t : \mathsf{S}\ l_{\mathtt{L}}\ \tau_1 \qquad \Gamma\,,\, x : \tau_1 \vdash_{\mathrm{nf}} s : \mathsf{S}\ l_{\mathtt{H}}\ \tau_2}{\Gamma \vdash_{\mathrm{nf}} \mathsf{let}{\uparrow}\ x = t\ \mathsf{in}\ s : \mathsf{S}\ l_{\mathtt{H}}\ \tau_2}$$

$$\textsc{Left}\ \frac{\Gamma \vdash_{\mathrm{nf}} t : \tau_1}{\Gamma \vdash_{\mathrm{nf}} \mathsf{left}\ t : \tau_1 + \tau_2} \qquad \textsc{Right}\ \frac{\Gamma \vdash_{\mathrm{nf}} t : \tau_2}{\Gamma \vdash_{\mathrm{nf}} \mathsf{right}\ t : \tau_1 + \tau_2}$$

$$\textsc{Case}\ \frac{\Gamma \vdash_{\mathrm{ne}} t : \tau_1 + \tau_2 \qquad \Gamma\,,\, x_1 : \tau_1 \vdash_{\mathrm{nf}} t_1 : \tau \qquad \Gamma\,,\, x_2 : \tau_2 \vdash_{\mathrm{nf}} t_2 : \tau}{\Gamma \vdash_{\mathrm{nf}} \mathsf{case}\ t\ (\mathsf{left}\ x_1 \to t_1)\ (\mathsf{right}\ x_2 \to t_2) : \tau}$$

Figure D.6.: Normal forms

Section D.5. The normal forms in Figure D.6 extend the β-short η-long forms in simply-typed λ-calculus (STLC) (Balat et al. 2004; Abel and Sattler 2019) with return and let↑. Note that, unlike neutrals, arbitrary normal forms do not obey neutrality since they may also construct values which do not occur in the context. For example, the normal form left () (which denotes the value *false*) of type $\emptyset \vdash_{\mathrm{nf}} \mathsf{Bool}$ constructs a value of the type Bool in the empty context $\emptyset$.

The reader may have noticed that the let↑ construct in normal forms does not directly resemble a term, and hence it is not immediately obvious how it should be quoted. Normal forms constructed by let↑ can be quoted by first applying up to the quotation of the neutral and then using let. The reason let↑ represents both let and up in the normal forms is to prevent reducibility of the normal forms. Had we added up separately to normal forms, then this may trigger further reductions. For example, the term up (return ()) can be reduced further to the term return (). Disallowing up-terms directly in normal forms removes the possibility of this reduction in normal forms. Similarly, adding up to neutral forms is also equally worse since it breaks neutrality.

The syntactic characterization of neutral and normal forms provides us with useful properties in the proof of noninterference. For example, there cannot exist a neutral of type $\emptyset \vdash_{\mathrm{ne}} \tau$ for any type τ. By neutrality, if such a neutral form exists, then τ must be a subformula of the empty context $\emptyset$, but this is impossible! Similarly, the η-long form of normal forms guarantee that a normal form of a function type must begin with either a λ or case—hence reducing the number of possible cases in our proof. In the next section, we utilize these properties to show that the program f (from earlier) behaves as a constant.

D.4. Normal Forms and Noninterference

The program $f : \mathsf{Int}_{\mathtt{H}} \to \mathsf{Bool}_{\mathtt{L}}$ from Section D.1 can be generalized in λ_{SEC} as a term[4] $\emptyset \vdash f : \mathsf{S}\ l_{\mathtt{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathtt{L}}\ \mathsf{Bool}$ marking the secret input and public output through the security monad. Noninterference for this term—which Russo et al. (2008) refer to as a "noninterference-like" property for λ_{SEC}—states that given two levels $l_{\mathtt{L}}$ (*public*) and $l_{\mathtt{H}}$ (*secret*) such that the flow of information from secret to public is disallowed as $l_{\mathtt{H}} \not\sqsubseteq l_{\mathtt{L}}$; for any two possibly different secrets s_1 and s_2, applying f to s_1 is equivalent to applying it to s_2. In other words, it states that varying the secret input must *not interfere* with the public output.

As explained before, for $\emptyset \vdash f : \mathsf{S}\ l_{\mathtt{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathtt{L}}\ \mathsf{Bool}$ to satisfy noninterference, it must be equivalent to the constant function whose body is return *true* or

[4] λ_{SEC} does not have polymorphic types, in this case τ represents an arbitrary but concrete type, for instance unit ().

return *false* independent of the input. For an arbitrary program f it is not possible to conclude so just from case analysis—as programs may be fairly complex—however, for normal forms of the same type it is possible. In the Lemma below, we materialize this intuition:

Lemma D.4.1 (Normal forms of f are constant)**.** *For any normal form* $\emptyset \vdash_{nf} f : \mathsf{S}\ l_{\mathsf{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}$*, either* $f \equiv \lambda\, x\,.\,(\mathsf{return}\ \mathit{true})$ *or* $f \equiv \lambda\, x\,.\,(\mathsf{return}\ \mathit{false})$

Note that the equality relation $\equiv$ denotes syntactic (or propositional) equality, which means that the normal forms on both sides must be syntactically identical. The proof follows by direct case analysis on the normal forms of type $\emptyset \vdash_{\mathrm{nf}} f : \mathsf{S}\ l_{\mathsf{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}$:

Proof of Lemma D.4.1. Upon closer inspection of the normal forms of λ_{SEC} (Figure D.6), the reader may notice that at function type $\emptyset \vdash_{\mathrm{nf}} \mathsf{S}\ l_{\mathsf{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}$ there exists only two possibilities: a case or a λ construct. The former, can be easily dismissed by neutrality because it requires the scrutinee—a neutral form of sum type $\tau_1 + \tau_2$—to appear in the empty context. In the latter case, the λ construct extends typing context of the body with the type of the argument, and thus refines the normal form to have the shape $\lambda\, x\,.\,\_$ where $\emptyset\ ,\ x : \mathsf{S}\ l_{\mathsf{H}}\ \tau \vdash_{\mathrm{nf}} \_ : \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}$.

Considering the normal forms of type $\emptyset\ ,\ x : \mathsf{S}\ l_{\mathsf{H}}\ \tau \vdash_{\mathrm{nf}} \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}$, we realize that there are only three possible candidates: the case construct again, the monadic return or let. As before, case is discharged because it requires the scrutinee of sum type to occur in the context $\emptyset\ ,\ x : \mathsf{S}\ l_{\mathsf{H}}\ \tau$. Analogously, the monadic let with a neutral term of type $\mathsf{S}\ l_{\mathsf{L}}\ \tau$, expects this type to occur in the same context—but it does not, since $\mathsf{S}\ l_{\mathsf{L}}\ \tau$ is not a subformula of $\mathsf{S}\ l_{\mathsf{H}}\ \tau$. The remaining case, return, can be further refined, where the only possibilities leave us with $\lambda\, x\,.\,(\mathsf{return}\ \mathit{true})$ or $\lambda\, x\,.\,(\mathsf{return}\ \mathit{false})$. □

In order to show that noninterference holds for arbitrary programs of type $\emptyset \vdash f : \mathsf{S}\ l_{\mathsf{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}$ using this lemma, we must link the behaviour of a program with that of its normal form. In the next section we develop the necessary normalization machinery and later complete the proof of noninterference in Section D.6.

D.5. From λ_{SEC} to Normal Forms

The goal of this section is to implement a normalization algorithm that bridges the gap between terms and their normal forms. For this purpose, we employ Normalization by Evaluation (NbE).

Normalization based on rewriting techniques (Pierce 2002) perform syntactic transformations of a term to produce a normal form. NbE, on the other hand, normalizes a term by evaluating it in a host language, and then extracting a normal form from the (semantic) value in the host language. Evaluation of a term is implemented by an interpreter function eval, and the extraction of normal forms, called *reification*, is implemented by an inverse function reify. Normalization is implemented as a function from terms to normal forms by composing these functions:

$$\begin{aligned}&\mathsf{norm} : (\Gamma \vdash \tau) \;\to\; (\Gamma \vdash_{\mathrm{nf}} \tau)\\ &\mathsf{norm}\ t \;=\; \mathsf{reify}\ (\mathsf{eval}\ t)\end{aligned}$$

The function eval and reify have the following types in the host language:

$$\begin{aligned}&\mathsf{eval} : (\Gamma \vdash \tau) \;\to\; (\llbracket\, \Gamma \,\rrbracket \;\to\; \llbracket\, \tau \,\rrbracket)\\ &\mathsf{reify} : (\llbracket\, \Gamma \,\rrbracket \;\to\; \llbracket\, \tau \,\rrbracket) \;\to\; (\Gamma \vdash_{\mathrm{nf}} \tau)\end{aligned}$$

In these types, the function $\llbracket\, \_ \,\rrbracket$ interprets types and contexts in λ_{SEC} as types in the host language. That is, the type $\llbracket\, \tau \,\rrbracket$ denotes the interpretation of the (λ_{SEC}) type τ in the host language, and similarly for $\llbracket\, \Gamma \,\rrbracket$. On the other hand, the function $\llbracket\, \Gamma \,\rrbracket \;\to\; \llbracket\, \tau \,\rrbracket$—a function between the interpretations in the host language—denotes the interpretation of the term $\Gamma \vdash \tau$.

The advantages of using NbE over a rewrite system are two-fold: first, it serves as an actual implementation of the normalization algorithm; second, and most importantly, when implemented in a proof system like Agda, it makes normalization amenable to formal reasoning. For example, since Agda ensures that all functions are total, we are assured that a normal form must exist for every term in λ_{SEC}. Similarly, we also get a proof that normalization terminates for free since Agda ensures that all functions are terminating.

We implement the functions eval and reify for terms in λ_{SEC} using Agda as the host language. Note that, however, the implementation of our algorithm—and NbE in general—is not specific to Agda. It may also be implemented in other programming languages such as Haskell (Danvy et al. 2001) or Standard ML (Balat et al. 2004).

In the remainder of this section, we will denote the typing derivations $\Gamma \vdash_{\mathrm{nf}} \tau$ and $\Gamma \vdash_{\mathrm{ne}} \tau$ as $\mathsf{Nf}\ \tau$ and $\mathsf{Ne}\ \tau$ respectively. We leave the context Γ implicit to avoid the clutter caused by contexts and their *weakenings* (Altenkirch et al. 1995; McBride 2018). Similarly, we will represent variables of type $\tau \in \Gamma$ as $\mathsf{Var}\ \tau$, leaving Γ implicit. Although we use de Bruijn indices in the actual implementation of variables, we will continue to use named variables here to ease presentation. We encourage the curious reader to see the formalization in Agda for further details.

D.5.1. NbE for Simple Types

To begin with, we implement evaluation and reification for the types ι, $()$, $\times$ and $\Rightarrow$. The implementation for sums is more technical, and hence deferred to Appendix D.I. Note that the implementation of NbE for simple types is entirely standard (Altenkirch et al. 1995; Balat et al. 2004). Their interpretation as Agda types is defined as follows:

$$\begin{array}{lcl} [\![\ \iota\ ]\!] & = & \mathsf{Nf}\ \iota \\ [\![\ ()\ ]\!] & = & \top \\ [\![\ \tau_1 \times \tau_2\ ]\!] & = & [\![\ \tau_1\ ]\!] \times [\![\ \tau_2\ ]\!] \\ [\![\ \tau_1 \Rightarrow \tau_2\ ]\!] & = & [\![\ \tau_1\ ]\!] \to [\![\ \tau_2\ ]\!] \end{array}$$

The types $()$, $\times$ and $\Rightarrow$ are simply interpreted as their counterparts in Agda. For the base type ι, however, we cannot provide a counterpart in Agda since we do not know anything about this type. Instead, since the type ι is not constructed or eliminated by any specific construct in λ_{SEC}, we simply require a normal form as an evidence for producing a value of type ι—and thus interpret it as $\mathsf{Nf}\ \iota$.

Typing contexts map variables to types, and hence their interpretation is an execution environment (or equivalently, a semantic substitution) defined like-wise:

$$\begin{array}{lcl} [\![\ \emptyset\ ]\!] & = & \emptyset \\ [\![\ \Gamma\ ,\ x : \tau_1\ ]\!] & = & [\![\ \Gamma\ ]\!]\ [\ \mathsf{Var}\ \tau_1 \mapsto [\![\ \tau_1\ ]\!]\ ] \end{array}$$

For example, a value γ which inhabits the interpretation $[\![\ \Gamma\ ]\!]$ denotes the execution environment for evaluating a term typed in the context Γ.

Given these definitions, evaluation is implemented as a straightforward interpreter function:

$$\begin{array}{lcl} \mathsf{eval}\ x & \gamma & = \mathsf{lookup}\ x\ \gamma \\ \mathsf{eval}\ () & \gamma & = \mathsf{tt} \\ \mathsf{eval}\ (\mathsf{fst}\ t) & \gamma & = \pi_1\ (\mathsf{eval}\ t\ \gamma) \\ \mathsf{eval}\ (\mathsf{snd}\ t) & \gamma & = \pi_2\ (\mathsf{eval}\ t\ \gamma) \\ \mathsf{eval}\ (< t_1\ ,\ t_2\ >) & \gamma & = (\mathsf{eval}\ t_1\ \gamma\ ,\ \mathsf{eval}\ t_2\ \gamma) \\ \mathsf{eval}\ (\lambda\ x\,.\,t) & \gamma & = \lambda\ v\ \to\ \mathsf{eval}\ t\ (\gamma\ [x \mapsto v]) \\ \mathsf{eval}\ (t\ s) & \gamma & = (\mathsf{eval}\ t\ \gamma)\ (\mathsf{eval}\ s\ \gamma) \end{array}$$

Note that γ is an execution environment for the term's context; lookup, π_1 and π_2 are Agda functions; and tt is the constructor of the unit type $\top$. For the case of $\lambda\ x\,.\,t$, evaluation is expected to return an equivalent semantic function.

We compute the body of this function by evaluating the body term t using the substitution γ extended with a mapping which assigns the value v to the variable x—denoted $\gamma\,[x \mapsto v]$.

Reification, on the other hand, is implemented using two helper functions reflect and reifyVal. The function reflect converts neutral forms to semantic values, while the dual function reifyVal converts semantic values to normal forms. These functions are implemented as follows:

```
reifyVal : ⟦ τ ⟧ → Nf τ
reifyVal {ι} n          = n
reifyVal {()} tt        = ()
reifyVal {τ₁ × τ₂} p    =
  < reifyVal {τ₁} (π₁ p) , reifyVal {τ₂} (π₁ p) >
reifyVal {τ₁ ⇒ τ₂} f    =
  λ x . reifyVal {τ₂} (f (reflect {τ₁} x))  |  fresh x


reflect : Ne τ → ⟦ τ ⟧
reflect {ι} n           = n
reflect {()} n          = tt
reflect {τ₁ × τ₂} n     =
  (reflect {τ₁} (fst n) , reflect {τ₂} (snd n))
reflect {τ₁ ⇒ τ₂} n     =
  λ v → reflect {τ₂} (n (reifyVal {τ₁} v))
```

Note that the argument inside the braces { } denotes an implicit parameter, which is the type of the corresponding neutral/value argument of reflect/reifyVal here.

Reflection is implemented by performing a type-directed translation of neutral forms to semantic values by induction on types. The interpretation of types, defined earlier, guides our implementation. For example, reflection of a neutral with a function type must produce a function value since the type $\Rightarrow$ is interpreted as an Agda function. For this purpose, we are given the argument value in the semantics and it remains to construct a function body of the appropriate type. We produce the body of this function by recursively reflecting a neutral application of the function and (the reification of) the argument value. The function reifyVal is also implemented in a similar fashion by induction on types.

To implement reification, recollect that the argument to reify is a function that results from partially applying the eval function with a term. If the term has type $\Gamma \vdash \tau$, then the argument, say f, must have the type $[\![\,\Gamma\,]\!] \to [\![\,\tau\,]\!]$.

Thus, to apply f, we need an execution environment of the type $[\![\ \Gamma\ ]\!]$. This environment can be generated by simply reflecting the variables in the context as follows:

$$\begin{aligned}
&\mathsf{genEnv} : (\Gamma : \mathsf{Ctx}) \ \to\ [\![\ \Gamma\ ]\!] \\
&\mathsf{genEnv}\ \emptyset \qquad\quad\ = \ \emptyset \\
&\mathsf{genEnv}\ (\Gamma\ ,\ x : \tau) \ = \ \mathsf{genEnv}\ \Gamma\ [\ x \mapsto \mathsf{reflect}\ x\ ]
\end{aligned}$$

Finally, we can now implement reify as follows:

$$\mathsf{reify}\ \{\Gamma\}\ f \ = \ \mathsf{let}\ \gamma \ = \ \mathsf{genEnv}\ \Gamma\ \mathsf{in}\ \mathsf{reifyVal}\ (f\ \gamma)$$

We generate an environment γ to apply the semantic function f, and then convert the resulting semantic value to a normal form by applying reifyVal.

D.5.2. NbE for the Security Monad

To interpret a type $\mathsf{S}\ l\ \tau$, we need a semantic counterpart in the host language which is also a monad. Suppose that we define such a monad as an inductive data type T parameterized by a label l and some type a (which would be $[\![\ \tau\ ]\!]$ in this case). Evidently this monad must allow the implementation of the semantic counterparts of the terms return, let and up in λ_{SEC} as follows:

$$\begin{aligned}
&\mathsf{return} : a \ \to\ T\ l\ a \\
&\mathsf{bind} \quad : T\ l\ a \ \to\ (a \ \to\ T\ l\ b) \ \to\ T\ l\ b \\
&\mathsf{up} \quad\ : (l_{\mathrm{L}} \ \sqsubseteq\ l_{\mathrm{H}}) \ \to\ T\ l_{\mathrm{L}}\ a \ \to\ T\ l_{\mathrm{H}}\ a
\end{aligned}$$

To satisfy this specification, we define the data type T in Agda with the following constructors:

$$\textsc{Return}\ \frac{x : a}{\mathsf{return}\ x : T\ l\ a} \qquad \textsc{BindN}\ \frac{p : l_{\mathrm{L}} \ \sqsubseteq\ l_{\mathrm{H}} \qquad n : \mathsf{Ne}\ \mathsf{S}\ l_{\mathrm{L}}\ \tau \qquad f : \mathsf{Var}\ \tau \ \to\ T\ l_{\mathrm{H}}\ a}{\mathsf{bindNe}\ p\ n\ f : T\ l_{\mathrm{H}}\ a}$$

The constructor return returns a semantic value in the monad, while bindNe registers a binding of a neutral to monadic value. These constructors are the semantic equivalent of return and let↑ in the normal forms, respectively. The constructor bindNe is more general than the required function bind in order to allow the definition of up, which is defined by induction as follows:

$$\begin{aligned}
&\mathsf{up}\ p\ (\mathsf{return}\ v) \ = \\
&\quad \mathsf{return}\ v
\end{aligned}$$

$$
\begin{aligned}
&\mathsf{up}\ p\ (\mathsf{bindNe}\ q\ n\ f)\ = \\
&\quad \mathsf{bindNe}\ (\mathsf{trans}\ q\ p)\ n\ (\lambda\ x\ \rightarrow\ \mathsf{up}\ p\ (f\ x))
\end{aligned}
$$

To understand this implementation, suppose that $p : l_{\mathtt{M}} \sqsubseteq l_{\mathtt{H}}$ for some labels $l_{\mathtt{M}}$ and $l_{\mathtt{H}}$. A monadic value of type $T\ l_{\mathtt{M}}\ a$ which is constructed by a return can be simply relabelled to $T\ l_{\mathtt{H}}\ a$ since return can be used to construct a monadic value on any label. For the case of $\mathsf{bindNe}\ q\ n\ f$, we have that $q : l_{\mathtt{L}} \sqsubseteq l_{\mathtt{M}}$ and $n : \mathsf{Ne}\ \mathsf{S}\ l_{\mathtt{L}}\ \tau_1$, hence $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$ by transitivity, and we may simply use bindNe to register n and recursively apply up on the continuation f to produce the desired result of type $T\ l_{\mathtt{H}}\ a$.

Using the type T in the host language, we may now interpret the monad in λ_{SEC} as follows:

$$
[\![\ \mathsf{S}\ l\ \tau\ ]\!]\ =\ T\ l\ [\![\ \tau\ ]\!]
$$

Having mirrored the monadic primitives in λ_{SEC} using semantic counterparts, evaluation is rather simple:

$$
\begin{aligned}
&\mathsf{eval}\ (\mathsf{return}\ t)\ \gamma\ =\ \mathsf{return}\ (\mathsf{eval}\ t\ \gamma) \\
&\mathsf{eval}\ (\mathsf{up}\ p\ t)\ \ \gamma\ =\ \mathsf{up}\ p\ (\mathsf{eval}\ t\ \gamma) \\
&\mathsf{eval}\ (\mathsf{let}\ x = t\ \mathsf{in}\ s)\ \gamma\ = \\
&\quad \mathsf{bind}\ (\mathsf{eval}\ t\ \gamma)\ (\lambda\ v\ \rightarrow\ \mathsf{eval}\ s\ (\gamma\ [x \mapsto v]))
\end{aligned}
$$

For implementing reflection, we can use bindNe to register a neutral binding and recursively reflect the given variable:

$$
\begin{aligned}
&\mathsf{reflect}\ \{\mathsf{S}\ l\ \tau\}\ n\ = \\
&\quad \mathsf{bindNe}\ \mathsf{refl}\ n\ (\lambda\ x\ \rightarrow\ \mathsf{return}\ (\mathsf{reflect}\ \{\tau\}\ x))
\end{aligned}
$$

Since we do not need to increase the sensitivity of the neutral to bind it here, we simply provide the "reflexive flow" $\mathsf{refl} : l \sqsubseteq l$.

The function reifyVal, on the other hand, is rather straightforward since the constructors of T are essentially semantic counterparts of the normal forms, and can hence be translated to it:

$$
\begin{aligned}
&\mathsf{reifyVal}\ \{\mathsf{S}\ l\ \tau\}\ (\mathsf{return}\ v)\ = \\
&\quad \mathsf{return}\ (\mathsf{reifyVal}\ \{\tau\}\ v) \\
&\mathsf{reifyVal}\ \{\mathsf{S}\ l\ \tau\}\ (\mathsf{bindNe}\ \{p\}\ n\ f)\ = \\
&\quad \mathsf{let}{\uparrow}\ \{p\}\ x = n\ \mathsf{in}\ \mathsf{reifyVal}\ \{\tau\}\ (f\ x)
\end{aligned}
$$

D.5.3. Preservation of Semantics

To prove that normalization preserves static semantics of λ_{SEC}, we must show that the normal form of term is equivalent to the term. Since normal forms and terms belong to different syntactic categories, we must first quote normal forms to state this relationship using the term equivalence relation $\approx$. This property, called *consistency* of normal forms, is stated as follows:

Theorem D.5.1 (Consistency of normal forms)**.** *For any term* $\Gamma \vdash t : \tau$ *we have that* $\Gamma \vdash t \approx \ulcorner \mathit{norm}\ t \urcorner : \tau$

An attempt to prove consistency by induction on the terms or types fails quickly since the induction principle alone is not strong enough to prove this theorem. To solve this issue we must establish a notion of equivalence between a term and its interpretation using *logical relations* (Plotkin 1980). Using these relations, we can prove that evaluation is consistent by showing that it is *related* to applying a substitution in the syntax. Following this, we can also prove the consistency of reification by showing that reifying a value related to a term, yields a normal form which is equivalent to the term when quoted. The consistency of evaluation and reification yields the proof of consistency for normal forms.

This proof follows the style of the consistency proof of NbE for STLC using Kripke logical relations by Coquand (1993). As is the case for sums, NbE for the security monad uses an inductively defined data type to implement the semantic monad. Hence, we are able to leverage the proof techniques used to prove the consistency of NbE for sums (Valliappan and Russo 2019) to prove the same for the security monad. We skip the details of the proof here, but encourage the curious reader to see the Agda mechanization of this theorem.

D.6. Noninterference for λ_{SEC}

After developing the necessary machinery to normalize terms in the calculus, we are ready to state and prove noninterference for λ_{SEC}. First, we complete the proof of noninterference for the program f from Section D.4.

D.6.1. Special Case of Noninterference

Theorem D.6.1 (Noninterference for f)**.** *Given security levels* l_{L} *and* l_{H} *such that* $l_{\text{H}} \not\sqsubseteq l_{\text{L}}$ *and a function* $\emptyset \vdash f : S\ l_{\text{H}}\ \tau \Rightarrow S\ l_{\text{L}}\ \mathit{Bool}$ *then* $\forall\ s_1\ s_2 : S\ l_{\text{H}}\ \tau.\ f\ s_1 \approx f\ s_2$

The proof of Theorem D.6.1 relies upon two key ingredients: Lemma D.4.1 (Section D.4), which characterizes the shape of the normal forms of f; and consistency of normal forms, Theorem D.5.1 (Section D.5.3), which links the semantics of f with that of its normal forms.

Proof of Theorem D.6.1. To show that a function $\emptyset \vdash f : \mathsf{S}\ l_{\mathtt{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathtt{L}}\ \mathsf{Bool}$ is equivalent when applied to two different secret inputs s_1 and s_2, first, we instantiate Lemma D.4.1 with the normal form of f, denoted by $\mathsf{norm}\ f$. In this manner, we obtain that the normal forms of f are exactly the constant function that returns *true* or *false* wrapped in the return. In the former case, by correctness of normalization we have that $f \approx \ulcorner \mathsf{norm}\ f \urcorner \approx \lambda\, x\,.\mathsf{return}\ \mathit{true}$. By β-reduction and congruence of term-level function application, we have that $\forall\ t.\ (\lambda\, x\,.\mathsf{return}\ \mathit{true})\ t \approx \mathsf{return}\ \mathit{true}$. Therefore, $f\ s_1 \approx f\ s_2$. The case when $\mathsf{norm}\ f \equiv \lambda\, x\,.\mathsf{return}\ \mathit{false}$ follows a similar argument. □

The noninterference property proven above characterizes what it means for a concrete class of programs, i.e. those of type $\emptyset \vdash f : \mathsf{S}\ l_{\mathtt{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathtt{L}}\ \mathsf{Bool}$, to be secure: the attacker cannot even learn one bit of the secret from using program f. Albeit interesting, this property does not scale to more complex programs; for instance if the function f was typed in a non empty context the proof of the above lemma would not hold. The rest of this section is dedicated to generalize and prove noninterference from the program f to arbitrary programs written in λ_{SEC}. As will become clear, normal forms of λ_{SEC} play a crucial role towards proving noninterference.

D.6.2. General Noninterference Theorem

In order to discuss general noninterference for λ_{SEC}, we must first specify what are the *secret* ($l_{\mathtt{H}}$) inputs of a program and its *public* ($l_{\mathtt{L}}$) output with respect to an attacker at level $l_{\mathtt{L}}$. The attacker can only learn information of a program by running it with different secret inputs and then observing its public output. Because the attacker can only observe outputs at their security level, we restrict the security condition to only consider programs where outputs are fully observable, i.e. *transparent* and *ground*, to the attacker.

Definition D.6.1 (Transparent type)**.**

- $()$ is transparent at any level l.
- ι is transparent at any level l.
- $\tau_1 \Rightarrow \tau_2$ is transparent at l iff τ_2 is transparent at l.

- $\tau_1 + \tau_2$ is transparent at l iff τ_1 and τ_2 are transparent at l.
- $\tau_1 \times \tau_2$ is transparent at l iff τ_1 and τ_2 are transparent at l.
- $\mathsf{S}\ l'\ \tau$ is transparent at l iff $l' \sqsubseteq l$ and τ is transparent at l.

Definition D.6.2 (Ground type)**.**

- $()$ is ground.
- ι is ground.
- $\tau_1 + \tau_2$ is ground iff τ_1 and τ_2 are ground.
- $\tau_1 \times \tau_2$ is ground iff τ_1 and τ_2 are ground.
- $\mathsf{S}\ l\ \tau$ is ground iff τ is ground.

A type τ is transparent at security level $l_{\mathbb{L}}$ if the type does not include the security monad type over a higher security level $l_{\mathbb{H}}$. A ground type, on the other hand, is a first order type, i.e. a type that does not contain a function type. These simplifying restrictions over the output type of a program allow us to state a generic noninterference property over terms and perform induction on the normal forms.

These restrictions do not hinder the generality of our security condition: a program producing a partially public output, for instance of product type $\mathsf{S}\ l_{\mathbb{L}}\ \mathsf{Bool} \times \mathsf{S}\ l_{\mathbb{H}}\ \mathsf{Bool}$, can be transformed to produce a fully public output by applying the snd projection. We return to this example later at the end of the section. Also note that previous work on proving noninterference for static IFC languages (Abadi et al. 1999; Miyamoto and Igarashi 2004) impose similar restrictions.

Departing from the traditional view of programs as closed terms, i.e. terms without free variables, in the λ_{SEC} calculus we consider all terms for which a typing derivation exists. This includes terms that contain free variables—unknowns—typed by the context, which we identify as the program inputs. Note that open terms are more general since they can always be closed as a function by abstracting over the free variables.

Now, we state what it means for a context to be secret at level l. These definitions, dubbed l-sensitivity, force the types appearing in the context to be at least as sensitive as l.

Definition D.6.3 (Context sensitivity)**.**

A context Γ is l-sensitive if and only if for all types $\tau \in \Gamma$, τ is l-sensitive. A type τ is l-sensitive, on the other hand, if and only if:

- τ is the function type $\tau_1 \Rightarrow \tau_2$ and τ_2 is l-sensitive.
- τ is the product type $\tau_1 \times \tau_2$ and τ_1 and τ_2 are l-sensitive.
- τ is the monadic type $\mathsf{S}\ l'\ \tau_1$ and $l \sqsubseteq l'$.

Next, we define substitutions[5], which lay at the core of β-reduction rules in the λ_{SEC} calculus. Substitutions map free variables in a term to other terms possibly typed in a different context.

$$\textit{Substitution}\ \sigma ::= \sigma_\emptyset \mid \sigma\ [x \mapsto t]$$

$$\boxed{\Gamma \vdash_{\text{sub}} \sigma : \Delta}$$

$$\text{(121)}\ \frac{\Gamma \vdash_{\text{sub}} \sigma : \Delta \qquad \Gamma \vdash t : \tau}{\Gamma \vdash_{\text{sub}} \sigma\ [\ x \mapsto t\ ] : \Delta\ ,\ x : \tau} \qquad \text{(122)}\ \frac{}{\Gamma \vdash_{\text{sub}} \sigma_\emptyset : \emptyset}$$

Figure D.7.: Substitutions for λ_{SEC}

A substitution is either empty, $\sigma_\emptyset$, or is the substitution σ extended with a new mapping from the variable $x : \tau$ to term t. We denote $t\ [\ \sigma\ ]$ the application of substitution σ to term t. Its definition is standard by induction on the term structure, thus we omit it here and refer the reader to the Agda formalization.

Substitutions, in general, provide a mix of terms of secret and public type to fill the variables in the context Γ of a program. However, for noninterference we need to fix the public part of the substitution and allow the secret part to vary. We do so by splitting a substitution σ into the composition of a public substitution, $\Gamma \vdash_{\text{sub}} \sigma_{l_{\mathbb{L}}} : \Delta$, that fixes the public inputs, and a secret substitution $\Delta \vdash_{\text{sub}} \sigma_{l_{\mathbb{H}}} : \Sigma$, that restricts Δ to be $l_{\mathbb{H}}$-sensitive. The composition of both, denoted $\Gamma \vdash_{\text{sub}} (\sigma_{l_{\mathbb{L}}}\ ;\ \sigma_{l_{\mathbb{H}}}) : \Sigma$, maps variables in context Γ to terms typed in Σ: first, $\sigma_{l_{\mathbb{L}}}$ maps variables from Γ to terms in Δ, subsequently, $\sigma_{l_{\mathbb{H}}}$ maps variables in Δ to terms typed in Σ. Below, we state $l_{\mathbb{L}}$-equivalence of substitutions:

Definition D.6.4 (Low equivalence of substitutions)**.**

Two substitutions σ_1 and σ_2 are $l_{\mathbb{L}}$-equivalent , written $\sigma_1 \approx_{l_{\mathbb{L}}} \sigma_2$, if and only if for all $l_{\mathbb{H}}$ such that $l_{\mathbb{H}} \not\sqsubseteq l_{\mathbb{L}}$, there exists a public substitution $\sigma_{l_{\mathbb{L}}}$, and two secret substitutions $\sigma^1_{l_{\mathbb{H}}}$ and $\sigma^2_{l_{\mathbb{H}}}$, such that $\sigma_1 \equiv \sigma_{l_{\mathbb{L}}}\ ;\ \sigma^1_{l_{\mathbb{H}}}$ and $\sigma_2 \equiv \sigma_{l_{\mathbb{L}}}\ ;\ \sigma^2_{l_{\mathbb{H}}}$

[5] In Section D.2 we purposely left capture-avoiding substitutions underspecified, we amend that here.

Informally, noninterference for λ_{SEC} states that applying two low equivalent substitutions to an arbitrary term whose type is ground and transparent yields two equivalent programs. As previously explained, intuitively a program satisfies such property if it is equivalent to a *constant* program: i.e. a program where the output does not depend on the input—in this case the variables in the typing context. As in Section D.4, instead of defining and proving this on arbitrary terms, we achieve this using normal forms.

Constant Terms and Normal Forms We prove the noninterference theorem by showing that terms of a type at level l_{L}, typed in a l_{H}-sensitive context, must be constant. We achieve this in turn by showing that the normal forms of such terms are constant. Below, we state when a term is constant:

Definition D.6.5 (Constant term)**.**
A term $\Gamma \vdash t : \tau$ is said to be constant if, for any two substitutions σ_1 and σ_2, we have that $t\ [\ \sigma_1\ ] \approx t\ [\ \sigma_2\ ]$.

Similarly, we must define what it means for a normal form to be constant. However, we cannot state this for normal forms directly using substitutions since the result of applying a substitution to a normal form may not be a normal form. For example, the result of substituting the variable x in the normal form $x : \iota \Rightarrow \iota\ ,\ y : \iota \vdash_{\text{nf}} x\ y : \iota$ by the identity function is not a normal form—and *cannot* be derived syntactically as a normal form using $\vdash_{\text{nf}}$. Instead, we lean on the shape of the context to state the property.

If a normal form $\Gamma \vdash_{\text{nf}} n : \tau$ is constant, then there must exist a syntactically identical derivation $\emptyset \vdash_{\text{nf}} n' : \tau$ such that $n \equiv n'$. However, since n and n' are typed in different contexts, Γ and $\emptyset$, it is not possible to compare them for syntactic equality. We solve this problem by *renaming* the normal form n' to add as many variables as mentioned in context Γ. The signature of the renaming function is the following:

$$\mathsf{ren} : \{\Gamma \leqslant \Delta\} \to (\Gamma \vdash_{\text{nf}} \tau) \to (\Delta \vdash_{\text{nf}} \tau)$$

The relation $\leqslant$ between contexts Γ and Δ indicates that the variables appearing in Δ are at least those present in Γ. This relation, called *weakening*, is defined as follows:

- $\emptyset \leqslant \emptyset$
- If $\Gamma \leqslant \Delta$, then $\Gamma \leqslant \Delta\ ,\ x : \tau$
- If $\Gamma \leqslant \Delta$, then $\Gamma\ ,\ x : \tau \leqslant \Delta\ ,\ x : \tau$

The function ren can be defined by simple induction on the derivation of the normal forms. Note that terms can also be renamed in the same fashion.

Definition D.6.6 (Constant normal form)**.** A normal form $\Gamma \vdash_{\text{nf}} n : \tau$ is constant if there exists a normal form $\emptyset \vdash_{\text{nf}} n' : \tau$ such that $\mathsf{ren}\ (n') \equiv n$.

Further, we need a lemma showing that if a term is constant, then so is its normal form.

Lemma D.6.1 (Constant plumbing lemma)**.** *If the normal form n of a term $\Gamma \vdash t : \tau$ is constant, then so is t.*

The proof follows by induction on the normal forms:

Proof of Lemma D.6.1. If n is constant, then there must exist a normal form $\emptyset \vdash_{\text{nf}} n' : \tau$ such that $\mathsf{ren}\ (n') \equiv n$. Let the quotation of this normal form $\ulcorner\ n'\ \urcorner$ be some term $\emptyset \vdash t' : \tau$. Recall from earlier that terms can also be renamed, hence we have $\mathsf{ren}\ (t') \approx \mathsf{ren}\ (\ulcorner\ n'\ \urcorner)$ by correctness of n'. Since it can be shown that $\mathsf{ren}\ (\ulcorner\ n'\ \urcorner) \equiv \ulcorner\ \mathsf{ren}\ (n')\ \urcorner$, we have that $\mathsf{ren}\ (\ulcorner\ n'\ \urcorner) \equiv \ulcorner\ n\ \urcorner$, and by correctness of n, we also have $\mathsf{ren}\ (t') \approx t$ — (1).

A substitution σ maps free variables in a term to terms. The empty substitution, denoted $\sigma_\emptyset$, is the unique substitution, such that $\Delta \vdash t' [\ \sigma_\emptyset\ ] : \tau$ for any Δ. That is, applying the empty substitution simply renames the term. We can show that $t' [\ \sigma_\emptyset\ ] \equiv \mathsf{ren}\ (t')$, and hence, by (1), we have $t' [\ \sigma_\emptyset\ ] \approx t$ — (2). Since $\sigma_\emptyset$ renames a term typed in the empty context, we can show that for any substitution σ, we have $(t' [\ \sigma_\emptyset\ ])\ [\ \sigma\ ] \approx t' [\ \sigma_\emptyset\ ]$. Because $\sigma_\emptyset$ is also unique, for any two substitutions σ_1 and σ_2, we have $(t' [\ \sigma_\emptyset\ ])\ [\ \sigma_1\ ] \approx (t' [\ \sigma_\emptyset\ ])\ [\ \sigma_2\ ]$ by transitivity of $\approx$. As a result, from (2), we achieve the desired result, $t [\ \sigma_1\ ] \approx t [\ \sigma_2\ ]$, therefore t must be constant. □

The key insight of our noninterference proof is reflected in the following lemma which shows how normal forms of λ_{SEC} typed in a sensitive context are either constant or the flow between the security level of the context and the output type is permitted. Below we include the proof to showcase how it follows by straightforward induction on the shape of the normal forms.

Lemma D.6.2 (Normal forms do not leak)**.** *Given a normal form $\Gamma \vdash_{nf} n : \tau$, where the context Γ is l_i-sensitive, and τ is a ground and transparent type at level l_o, then either n is constant or $l_i \sqsubseteq l_o$.*

Proof. By induction on the structure of the normal form n. Note that λ and case normal forms need not be considered since the preconditions ensure that τ cannot be a function type (dismisses λ), and Γ cannot contain a variable of a sum type (dismisses case).

- **Case 1** ($\Gamma \vdash_{\text{nf}} () : ()$). The normal form $()$ is constant.
- **Case 2** ($\Gamma \vdash_{\text{nf}} n : \iota$). In this case, we are given the neutral n by the [Base] rule in Figure D.6. It can be shown by induction that for all neutrals of type $\Gamma \vdash_{\text{ne}} \tau$, if Γ is l_i-sensitive and τ is transparent at l_o, then $l_i \sqsubseteq l_o$. Hence, n gives us that $l_i \sqsubseteq l_o$.
- **Case 3** ($\Gamma \vdash_{\text{nf}} \mathsf{return}\ n : \mathsf{S}\ l\ \tau$). By applying the induction hypothesis on the normal form n, we have that n is either constant or $l_i \sqsubseteq l_o$. In the latter case, we are done since we already have $l_i \sqsubseteq l_o$. In the former case, there exists a normal form n' such that $\mathsf{ren}\ (n') \equiv n$. By congruence of the relation $\equiv$, we get that $\mathsf{return}\ (\mathsf{ren}\ (n')) \equiv \mathsf{return}\ n$. Note that the function ren is defined as $\mathsf{ren}\ (\mathsf{return}\ n') \equiv \mathsf{return}\ (\mathsf{ren}\ n')$, and hence by transitivity of $\equiv$, we have that $\mathsf{ren}\ (\mathsf{return}\ (n')) \equiv \mathsf{return}\ n$. Thus, the normal form $\mathsf{return}\ n$ is also constant.
- **Case 4** ($\Gamma \vdash_{\text{nf}} \mathsf{let}{\uparrow}\ x = n\ \mathsf{in}\ m : \mathsf{S}\ l_2\ \tau_2$). For this case, we have a neutral $\Gamma \vdash_{\text{ne}} n : \mathsf{S}\ l_1\ \tau_1$ such that $l_1 \sqsubseteq l_2$, by the [LetUp] rule in Figure D.6. Similar to case 2, we have that $l_i \sqsubseteq l_1$ from the neutral n. Hence, $l_i \sqsubseteq l_2$ by transitivity of the relation $\sqsubseteq$. Additionally, since $\mathsf{S}\ l_2\ \tau$ is transparent at l_o, it must be the case that $l_2 \sqsubseteq l_o$ by definition of transparency. Therefore, once again by transitivity, we have $l_i \sqsubseteq l_o$.
- **Case 5** ($\Gamma \vdash_{\text{nf}} \mathsf{left}\ n : \tau_1 + \tau_2$). Similar to return.
- **Case 6** ($\Gamma \vdash_{\text{nf}} \mathsf{right}\ n : \tau_1 + \tau_2$). Similar to return.

□

The last step to noninterference is an ancillary lemma which shows that terms typed in l_{H}-sensitive contexts are constant:

Lemma D.6.3. *Given a term $\Gamma \vdash t : \tau$, where the context Γ is l_{H}-sensitive, and τ is a ground type transparent at l_{L}. If $l_{\text{H}} \not\sqsubseteq l_{\text{L}}$, then t is constant.*

The proof follows from Lemmas D.6.2 and D.6.1.

Finally, we are ready to formally state and prove the noninterference property for programs written in λ_{SEC}, which effectively demonstrates that programs do not leak sensitive information. The proof follows from the previous lemmas, which characterize the behaviour of programs by the syntactic properties of their normal forms.

Theorem D.6.2 (Noninterference for λ_{SEC})**.** *Given security levels l_{L} and l_{H} such that $l_{\mathsf{H}} \not\sqsubseteq l_{\mathsf{L}}$; an attacker at level l_{L}; two l_{L}-equivalent substitutions σ_1 and σ_2 such that $\sigma_1 \approx_{l_{\mathsf{L}}} \sigma_2$; and a type τ that is ground and transparent at l_{L}; then for any term $\Gamma \vdash t : \tau$ we have that $t\,[\,\sigma_1\,] \approx t\,[\,\sigma_2\,]$.*

Proof of Theorem D.6.2. Low equivalence of substitutions $\sigma_1 \approx_{l_{\mathsf{L}}} \sigma_2$ gives that $\sigma_1 = \sigma_{l_{\mathsf{L}}} ; \sigma^1_{l_{\mathsf{H}}}$ and $\sigma_2 = \sigma_{l_{\mathsf{L}}} ; \sigma^2_{l_{\mathsf{H}}}$. After applying the public substitution $\sigma_{l_{\mathsf{L}}}$ to the term $\Gamma \vdash t : \tau$, we are left with a term typed in a l_{H}-sensitive context Δ, $\Delta \vdash t\,[\,\sigma_{l_{\mathsf{L}}}\,] : \tau$. By Lemma D.6.3, $t\,[\,\sigma_{l_{\mathsf{L}}}\,]$ is constant which means that $(t\,[\,\sigma_{l_{\mathsf{L}}}\,])\,[\,\sigma^1_{l_{\mathsf{H}}}\,] \approx (t\,[\,\sigma_{l_{\mathsf{L}}}\,])\,[\,\sigma^2_{l_{\mathsf{H}}}\,]$. By readjusting substitutions using composition we obtain $t\,([\,\sigma_{l_{\mathsf{L}}} ; \sigma^1_{l_{\mathsf{H}}}\,]) \approx t\,([\,\sigma_{l_{\mathsf{L}}} ; \sigma^2_{l_{\mathsf{H}}}\,])$, which yields $t\,[\,\sigma_1\,] \approx t\,[\,\sigma_2\,]$. □

D.6.3. Follow-up Example

To conclude this section, we briefly show how to instantiate the theorem of noninterference for λ_{SEC} for programs of type $\emptyset \vdash t : \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool} \times \mathsf{S}\ l_{\mathsf{H}}\ \mathsf{Bool} \Rightarrow \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool} \times \mathsf{S}\ l_{\mathsf{H}}\ \mathsf{Bool}$, which are the recurring example for explaining noninterference in the literature (Russo et al. 2008; Bowman and Ahmed 2015). Adapted to the notion of noninterference based on substitutions, the corollary we aim to prove is the following:

Corollary D.6.1 (Noninterference for t)**.** *Given security levels l_{L} and l_{H} such that $l_{\mathsf{H}} \not\sqsubseteq l_{\mathsf{L}}$ and a program $x : \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool} \times \mathsf{S}\ l_{\mathsf{H}}\ \mathsf{Bool} \vdash t : \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool} \times \mathsf{S}\ l_{\mathsf{H}}\ \mathsf{Bool}$ then $\forall\ p : \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}\ ,\ s_1\ s_2 : \mathsf{S}\ l_{\mathsf{H}}\ \mathsf{Bool}$. we have that $t\,[\,x \mapsto (p\ ,\ s_1)\,] \approx t\,[\,x \mapsto (p\ ,\ s_2)\,]$.*

Because the main noninterference theorem requires the output to be fully observable by the attacker, we transform t to the desired shape by applying the snd projection. This is justified because the first component of the output is protected at level l_{H}, which the attacker cannot observe. Below we prove noninterference for $x : \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool} \times \mathsf{S}\ l_{\mathsf{H}}\ \mathsf{Bool} \vdash \mathsf{snd}\ t : \mathsf{S}\ l_{\mathsf{H}}\ \mathsf{Bool}$:

Proof of Corollary D.6.1. To apply Theorem D.6.2 we have to show that both substitutions are low equivalent, $[\,x \mapsto (p\ ,\ s_1)\,] \approx_{l_{\mathsf{L}}} [\,x \mapsto (p\ ,\ s_2)\,]$ The key idea is that the substitution $[\,x \mapsto (p\ ,\ s_1)\,]$ can be decomposed into a public substitution $\sigma_{l_{\mathsf{L}}} \equiv [\,x \mapsto (p\ ,\ y)\,]$ and two different secret substitutions where each replaces the variable y by a different secret, $\sigma^1_{l_{\mathsf{H}}} \equiv [\,y \mapsto s_1\,]$ and $\sigma^2_{l_{\mathsf{H}}} \equiv [\,y \mapsto s_2\,]$. Now, the proof follows directly from Theorem D.6.2. □

D.7. Conclusions and Future Work

In this paper we have presented a novel proof of noninterference for the λ_{SEC} calculus (based on Haskell's IFC library SecLib) using normalization. The simplicity of the proof relies upon the normal forms of the calculus, which as opposed to arbitrary terms, are well-principled. To obtain normal forms from terms, we have implemented normalization using normalization by evaluation (NbE), and shown that normal forms obey useful syntactic properties such as neutrality and $\beta\eta$-long form. Most of the auxiliary lemmas and definitions towards proving noninterference build on these properties. Because normal forms are well-principled, many cases of the proofs follow directly by structural induction.

An important difference between our work and previous proofs based on term erasure is that our proof utilizes the static semantics of the language instead of the dynamic semantics. Specifically, our proof of noninterference is not tied to any particular evaluation strategy, such as call-by-name or call-by-value, assuming the strategy is adequate with respect to the static semantics.

Perhaps the closest to our line of work is the proof of noninterference by Miyamoto and Igarashi (2004) for a modal lambda calculus using normalization. The main novelty of our proof is that it works for standard extensions of the simply typed lambda calculus and does not change the typing rules of the underlying calculus (as presented and implemented by Russo et al. (2008)). This makes our proof technique applicable even in the presence of other useful normalization-preserving extensions of STLC. For example, it should be possible to extend our proof for λ_{SEC} further with exceptions and other *computational* effects (à la Moggi (1989)) since our security monad is already an instance of this. Moreover, our proof relies on syntactic properties of normal forms in an open typing context since normalization is based on the static semantics of the language.

In this work we have only considered a calculus which models terminating computations. This opens up a question of whether our proof technique is applicable to languages which support general recursion, where computations need not necessarily terminate. The extensibility of this technique to recursion relies directly upon the choice of static semantics for normalizing recursion. For example, it may be possible to extend the proof for λ_{SEC} with a fixpoint combinator by treating it as an uninterpreted constant during normalization. That is, it may be sufficient to normalize the body of the function by ignoring the recursive application, because if the body does not leak a secret, then its recursive call must not either. Since complete normalization is not strictly needed for our purposes, we believe that our technique can also be extended to

general recursion.

Our NbE implementation for λ_{SEC} extends NbE for Moggi's computational metalanguage (Filinski 2001; Lindley 2005) with a family of monads parameterized by a preordered set of labels. This resembles the parameterization of monads by effects specified by a preordered monoid, also known as *graded monads* (Wadler and Thiemann 2003; Orchard and Petricek 2014), and thus indicates the extensibility of our NbE algorithm to calculi with graded monads. It would be interesting to see if our proof technique can be used to prove noninterference for static enforcement of IFC using graded monads.

Using static semantics means that our work lays a foundation for static analysis of noninterference-like security properties. This opens up a plethora of exciting opportunities for future work. For example, one possibility would be to use type-directed partial evaluation (Danvy 1998) to simplify programs and inspect the resulting programs to verify if they violate security properties. Another arena would be the extension of our proof to more expressive IFC calculi such as dependency core calculus (DCC) or MAC (Vassena, Russo, et al. 2018). The main challenge here would be to identify the appropriate static semantics of the language, as they may not always have been designed with one in mind.

Acknowledgements

We thank Alejandro Russo, Fabian Ruch, Sandro Stucki and Maximilian Algehed for the insightful discussions on normalization and noninterference. We would also like to thank Irene Lobo Valbuena, Claudio Agustin Mista and the anonymous reviewers at PLAS'19 for their comments on earlier drafts of this paper. This work was funded by the Swedish Foundation for Strategic Research (SSF) under the projects WebSec (Ref. RIT17-0011) and Octopi (Ref. RIT17-0023).

Bibliography

Abadi, Martín et al. (1999). "A Core Calculus of Dependency". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, pp. 147–160. DOI: 10.1145/292540.292555. URL: https://doi.org/10.1145/292540.292555 (cit. on pp. 155, 172).

Abel, Andreas and Christian Sattler (2019). "Normalization by Evaluation for Call-By-Push-Value and Polarized Lambda Calculus". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by Ekaterina Komendantskaya. ACM, 3:1–3:12. DOI: 10.1145/3354166.3354168. URL: https://doi.org/10.1145/3354166.3354168 (cit. on pp. 159, 163, 184).

Algehed, Maximilian (2018). "A Perspective on the Dependency Core Calculus". In: *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by Mário S. Alvim and Stéphanie Delaune. ACM, pp. 24–28. DOI: 10.1145/3264820.3264823. URL: https://doi.org/10.1145/3264820.3264823 (cit. on p. 155).

Altenkirch, Thorsten et al. (1995). "Categorical Reconstruction of a Reduction Free Normalization Proof". In: *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings*. Ed. by David H. Pitt et al. Vol. 953. Lecture Notes in Computer Science. Springer, pp. 182–199. DOI: 10.1007/3-540-60164-3_27. URL: https://doi.org/10.1007/3-540-60164-3%5C_27 (cit. on pp. 165, 166).

Balat, Vincent et al. (2004). "Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Ed. by Neil D. Jones and Xavier Leroy. ACM, pp. 64–76. DOI: 10.1145/964001.964007. URL: https://doi.org/10.1145/964001.964007 (cit. on pp. 163, 165, 166).

Berger, Ulrich and Helmut Schwichtenberg (1991). "An Inverse of the Evaluation Functional for Typed lambda-calculus". In: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*. IEEE Computer Society, pp. 203–211. DOI: 10.1109/LICS.1991.151645. URL: https://doi.org/10.1109/LICS.1991.151645 (cit. on p. 156).

Bowman, William J. and Amal Ahmed (2015). "Noninterference for free". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, pp. 101–113. DOI: 10.1145/2784731.2784733. URL: https://doi.org/10.1145/2784731.2784733 (cit. on pp. 155, 177).

Coquand, Catarina (1993). "From Semantics to Rules: A Machine Assisted Analysis". In: *Computer Science Logic, 7th Workshop, CSL '93, Swansea, United Kingdom, September 13-17, 1993, Selected Papers*. Ed. by Egon Börger

et al. Vol. 832. Lecture Notes in Computer Science. Springer, pp. 91–105. DOI: 10.1007/BFb0049326. URL: https://doi.org/10.1007/BFb0049326 (cit. on p. 170).

Danvy, Olivier (1998). "Type-Directed Partial Evaluation". In: *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998*. Ed. by John Hatcliff et al. Vol. 1706. Lecture Notes in Computer Science. Springer, pp. 367–411. DOI: 10.1007/3-540-47018-2_16. URL: https://doi.org/10.1007/3-540-47018-2%5C_16 (cit. on p. 179).

Danvy, Olivier et al. (2001). "Normalization by evaluation with typed abstract syntax". In: *J. Funct. Program.* 11.6, pp. 673–680. DOI: 10.1017/S0956796801004166. URL: https://doi.org/10.1017/S0956796801004166 (cit. on p. 165).

Denning, Dorothy E. (1976). "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5, pp. 236–243. DOI: 10.1145/360051.360056. URL: https://doi.org/10.1145/360051.360056 (cit. on p. 157).

Filinski, Andrzej (2001). "Normalization by Evaluation for the Computational Lambda-Calculus". In: *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings*. Ed. by Samson Abramsky. Vol. 2044. Lecture Notes in Computer Science. Springer, pp. 151–165. DOI: 10.1007/3-540-45413-6_15. URL: https://doi.org/10.1007/3-540-45413-6%5C_15 (cit. on p. 179).

Kavvos, G. A. (2019). "Modalities, cohesion, and information flow". In: *Proc. ACM Program. Lang.* 3.POPL, 20:1–20:29. DOI: 10.1145/3290333. URL: https://doi.org/10.1145/3290333 (cit. on p. 155).

Li, Peng and Steve Zdancewic (2010). "Arrows for secure information flow". In: *Theor. Comput. Sci.* 411.19, pp. 1974–1994. DOI: 10.1016/j.tcs.2010.01.025. URL: https://doi.org/10.1016/j.tcs.2010.01.025 (cit. on p. 155).

Lindley, Sam (2005). "Normalisation by evaluation in the compilation of typed functional programming languages". PhD thesis. University of Edinburgh, UK. URL: https://hdl.handle.net/1842/778 (cit. on pp. 155, 159, 179).

McBride, Conor (2018). "Everybody's Got To Be Somewhere". In: *Proceedings of the 7th Workshop on Mathematically Structured Functional Programming, MSFP@FSCD 2018, Oxford, UK, 8th July 2018*. Ed. by Robert Atkey and Sam Lindley. Vol. 275. EPTCS, pp. 53–69. DOI: 10.4204/EPTCS.275.6. URL: https://doi.org/10.4204/EPTCS.275.6 (cit. on p. 165).

Miyamoto, Kenji and Atsushi Igarashi (2004). "A modal foundation for secure information flow". In: *In Proceedings of IEEE Foundations of Computer Security (FCS)*, pp. 187–203 (cit. on pp. 172, 178).

Moggi, Eugenio (1989). "Computational Lambda-Calculus and Monads". In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, pp. 14–23. DOI: 10.1109/LICS.1989.39155. URL: https://doi.org/10.1109/LICS.1989.39155 (cit. on pp. 157, 178).

– (1991). "Notions of Computation and Monads". In: *Inf. Comput.* 93.1, pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4. URL: https://doi.org/10.1016/0890-5401(91)90052-4 (cit. on pp. 156, 159).

Orchard, Dominic A. and Tomas Petricek (2014). "Embedding effect systems in Haskell". In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. Ed. by Wouter Swierstra. ACM, pp. 13–24. DOI: 10.1145/2633357.2633368. URL: https://doi.org/10.1145/2633357.2633368 (cit. on p. 179).

Pierce, Benjamin C. (2002). *Types and programming languages*. MIT Press. ISBN: 978-0-262-16209-8 (cit. on pp. 158, 165).

Plotkin, Gordon D. (1980). "Lambda-definability in the full type hierarchy". In: *To H. B. Curry: essays on combinatory logic, lambda calculus and formalism*. Academic Press, London-New York, pp. 363–373 (cit. on p. 170).

Russo, Alejandro et al. (2008). "A library for light-weight information-flow security in haskell". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, pp. 13–24. DOI: 10.1145/1411286.1411289. URL: https://doi.org/10.1145/1411286.1411289 (cit. on pp. 155, 156, 163, 177, 178).

Stefan, Deian et al. (2011). "Flexible dynamic information flow control in Haskell". In: *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. Ed. by Koen Claessen. ACM, pp. 95–106. DOI: 10.1145/2034675.2034688. URL: https://doi.org/10.1145/2034675.2034688 (cit. on pp. 155, 157).

Terei, David et al. (2012). "Safe haskell". In: *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*. Ed. by Janis Voigtländer. ACM, pp. 137–148. DOI: 10.1145/2364506.2364524. URL: https://doi.org/10.1145/2364506.2364524 (cit. on p. 157).

Tomé Cortiñas, Carlos and Nachiappan Valliappan (2019). "Simple Noninterference by Normalization". In: *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, CCS 2019, London, United Kingdom, November 11-15, 2019*. Ed. by Piotr Mardziel and Niki Vazou. ACM, pp. 61–72. DOI: 10.1145/3338504.3357342. URL: https://doi.org/10.1145/3338504.3357342 (cit. on p. 153).

Troelstra, Anne Sjerp and Helmut Schwichtenberg (2000). *Basic proof theory, Second Edition*. Vol. 43. Cambridge tracts in theoretical computer science. Cambridge University Press. ISBN: 978-0-521-77911-1 (cit. on p. 161).

Tse, Stephen and Steve Zdancewic (2004). "Translating dependency into parametricity". In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*. Ed. by Chris Okasaki and Kathleen Fisher. ACM, pp. 115–125. DOI: 10.1145/1016850.1016868. URL: https://doi.org/10.1145/1016850.1016868 (cit. on p. 155).

Valliappan, Nachiappan and Alejandro Russo (2019). "Exponential Elimination for Bicartesian Closed Categorical Combinators". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by Ekaterina Komendantskaya. ACM, 20:1–20:13. DOI: 10.1145/3354166.3354185. URL: https://doi.org/10.1145/3354166.3354185 (cit. on p. 170).

Vassena, Marco and Alejandro Russo (2016). "On Formalizing Information-Flow Control Libraries". In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*. Ed. by Toby C. Murray and Deian Stefan. ACM, pp. 15–28. DOI: 10.1145/2993600.2993608. URL: https://doi.org/10.1145/2993600.2993608 (cit. on p. 155).

Vassena, Marco, Alejandro Russo, et al. (2018). "MAC A verified static information-flow control library". In: *Journal of Logical and Algebraic Methods in Programming* 95, pp. 148–180. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2017.12.003. URL: https://www.sciencedirect.com/science/article/pii/S235222081730069X (cit. on pp. 157, 179).

Wadler, Philip and Peter Thiemann (2003). "The marriage of effects and monads". In: *ACM Trans. Comput. Log.* 4.1, pp. 1–32. DOI: 10.1145/601775.601776. URL: https://doi.org/10.1145/601775.601776 (cit. on p. 179).

Appendices

D.I. NbE for Sums

It is tempting to interpret sums component-wise like products and functions as: $[\![\tau_1 + \tau_2]\!] = [\![\tau_1]\!] \uplus [\![\tau_2]\!]$. However, this interpretation makes it impossible to implement reflection faithfully: should the reflection of a variable $x : \tau_1 + \tau_2$ be a semantic value of type $[\![\tau_1]\!]$ (left injection) or $[\![\tau_2]\!]$ (right injection)? We cannot make this decision since the value which substitutes x may be either of these cases. The standard solution to this issue is to interpret sums using

decision trees (Abel and Sattler 2019). A decision tree allows us to defer this decision until more information is available about the injection of the actual value.

As in the previous case for the monadic type T, a decision tree can be defined as an inductive data type D parameterized by some type interpretation a with the following constructors:

$$\frac{x : a}{\mathsf{leaf}\ x : D\ a}\ \textsc{Leaf} \qquad \frac{n : \mathsf{Ne}\ (\tau_1 + \tau_2) \qquad f : \mathsf{Var}\ \tau_1 \to D\ a \qquad g : \mathsf{Var}\ \tau_2 \to D\ a}{\mathsf{branch}\ n\ f\ g : D\ a}\ \textsc{Branch}$$

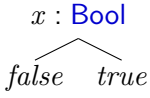
The leaf constructor constructs a leaf of the tree from a semantic value, while the branch constructor constructs a tree which represents a suspended decision over the value of a sum type. The branch constructor is the semantic equivalent of case in normal forms.

Decision trees allow us to model semantic sum values, and hence allow the interpretation of the sum type as follows:

$$[\![\ \tau_1 + \tau_2\ ]\!] = D\ ([\![\ \tau_1\ ]\!] \uplus [\![\ \tau_2\ ]\!])$$

We interpret a sum type (in λ_{SEC}) as a decision tree which contains a value of the sum type (in Agda).

As an example, the term *false* of type Bool, implemented as left (), will be interpreted as a decision tree leaf (inj_1 tt) of type $D\ [\![\ \mathsf{Bool}\ ]\!]$ since we know the exact injection. The Agda constructor inj_1 denotes the left injection in Agda, and inj_2 the right injection. For a variable x of type Bool, however, we cannot interpret it as a leaf since we don't know the actual injection that may substitute it. Instead, it is interpreted as a decision tree by branching over the possible values as branch x (λ _ $\to$ leaf (inj_1 tt)) (λ _ $\to$ leaf (inj_2 tt))[6]—which intuitively represents the following tree:



In light of this interpretation of sums, the implementation of evaluation for injections is straightforward since we only need to wrap the appropriate injection inside a leaf:

$$\begin{aligned}\mathsf{eval}\ (\mathsf{left}\ t)\ \ \gamma &= \mathsf{leaf}\ (\mathsf{inj}_1\ (\mathsf{eval}\ t\ \gamma))\\ \mathsf{eval}\ (\mathsf{right}\ t)\ \gamma &= \mathsf{leaf}\ (\mathsf{inj}_2\ (\mathsf{eval}\ t\ \gamma))\end{aligned}$$

[6]We ignore the argument (as λ _) here since it has the uninteresting type ()

For evaluating case however, we must first implement a decision procedure since case is used to make a choice over sums.

To make a decision over a tree of type $D\ [\![\ \tau\ ]\!]$, we need a function $\mathsf{mkDec} : D\ [\![\ \tau\ ]\!] \to [\![\ \tau\ ]\!]$. It can be implemented by induction on the type τ using monadic functions fmap and join on trees, which can in turn be implemented by straightforward structural induction on the tree. Additionally, we will also need a function which converts a decision over normal forms to a normal form: $\mathsf{convert} : D\ (\mathsf{Nf}\ \tau) \to \mathsf{Nf}\ \tau$. The implementation of this function is made possible by the fact that branch resembles case in normal forms, and can hence be translated to it. We skip the implementation of these functions here, but encourage the reader to see the Agda implementation.

Using these definitions, we can now complete evaluation as follows:

$$
\begin{aligned}
&\mathsf{eval}\ (\mathsf{case}\ t\ (\mathsf{left}\ x_1 \to t_1)\ (\mathsf{right}\ x_2 \to t_2))\ \gamma\ = \\
&\quad \mathsf{mkDec}\ (\mathsf{fmap}\ \mathsf{match}\ (\mathsf{eval}\ t\ \gamma)) \\
&\quad \mathsf{where} \\
&\qquad \mathsf{match} : ([\![\ \tau_1\ ]\!] \uplus [\![\ \tau_2\ ]\!]) \to [\![\ \tau\ ]\!] \\
&\qquad \mathsf{match}\ (\mathsf{inj}_1\ v)\ = \ \mathsf{eval}\ t_1\ (\gamma\ [x_1 \mapsto v]) \\
&\qquad \mathsf{match}\ (\mathsf{inj}_2\ v)\ = \ \mathsf{eval}\ t_2\ (\gamma\ [x_2 \mapsto v])
\end{aligned}
$$

We first evaluate the term t of type $\tau_1 + \tau_2$ to obtain a tree of type $D\ ([\![\ \tau_1\ ]\!] \uplus [\![\ \tau_2\ ]\!])$. Then, we map the function match which eliminates the sum inside the decision tree to $[\![\ \tau\ ]\!]$, to produce a tree of type $D\ [\![\ \tau\ ]\!]$. Finally, we run the decision procedure mkDec on the resulting decision tree to produce the desired value of type $[\![\ \tau\ ]\!]$.

Reflection for a neutral of a sum type can now be implemented using branch as follows:

$$
\begin{aligned}
&\mathsf{reflect}\ \{\tau_1 + \tau_2\}\ n\ = \\
&\quad \mathsf{branch}\ n \\
&\qquad (\mathsf{leaf}\ (\lambda\ x_1\ \to\ \mathsf{inj}_1\ (\mathsf{reflect}\ \{\tau_1\}\ x_1))) \\
&\qquad (\mathsf{leaf}\ (\lambda\ x_2\ \to\ \mathsf{inj}_2\ (\mathsf{reflect}\ \{\tau_2\}\ x_2)))
\end{aligned}
$$

As discussed earlier, we construct the decision tree for neutral n using branch. The subtrees represent all possible semantic values of n and are constructed by reflecting the variables x_1 and x_2.

The function reifyVal, on the other hand, is implemented similar to evaluation by eliminating the sum value inside the decision tree into normal forms as follows:

$$
\begin{aligned}
&\mathsf{reifyVal}\ \{\tau_1 + \tau_2\}\ tr\ = \ \mathsf{convert}\ (\mathsf{fmap}\ \mathsf{matchNf}\ tr) \\
&\quad \mathsf{where}
\end{aligned}
$$

```
matchNf : (⟦ τ1 ⟧ + ⟦ τ2 ⟧) → Nf (τ1 + τ2)
matchNf (inj1 x) = left (reifyVal {τ1} x)
matchNf (inj2 y) = right (reifyVal {τ2} y)
```

With this function, we have completed the implementation of NbE for sums.

E

Normalization for Fitch-Style Modal Calculi

Nachiappan Valliappan, Fabian Ruch, and Carlos Tomé Cortiñas

Proceedings of the ACM on Programming Languages Vol 6. ICFP 2022

Abstract Fitch-style modal lambda calculi enable programming with necessity modalities in a typed lambda calculus by extending the typing context with a delimiting operator that is denoted by a lock. The addition of locks simplifies the formulation of typing rules for calculi that incorporate different modal axioms, but each variant demands different, tedious and seemingly ad hoc syntactic lemmas to prove normalization. In this work, we take a semantic approach to normalization, called normalization by evaluation (NbE), by leveraging the possible-world semantics of Fitch-style calculi to yield a more modular approach to normalization. We show that NbE models can be constructed for calculi that incorporate the K, T and 4 axioms of modal logic, as suitable instantiations of the possible-world semantics. In addition to existing results that handle β-equivalence, our normalization result also considers η-equivalence for these calculi. Our key results have been mechanized in the proof assistant Agda. Finally, we showcase several consequences of normalization for proving meta-theoretic properties of Fitch-style calculi as well as programming-language applications based on different interpretations of the necessity modality.

E.1. Introduction

In type systems, a *modality* can be broadly construed as a unary type constructor with certain properties. Type systems with modalities have found a wide range of applications in programming languages to specify properties of a program in its type. In this work, we study typed lambda calculi equipped with a *necessity* modality (denoted by $\Box$) formulated in the so-called Fitch style.

The necessity modality originates from modal logic, where the most basic intuitionistic modal logic IK (for "intuitionistic" and "Kripke") extends intuitionistic propositional logic with a unary connective $\Box$, the *necessitation rule* (if $\cdot \vdash A$ then $\Gamma \vdash \Box A$) and the *K axiom* ($\Box(A \Rightarrow B) \Rightarrow \Box A \Rightarrow \Box B$). With the addition of further modal axioms T ($\Box A \Rightarrow A$) and 4 ($\Box A \Rightarrow \Box\Box A$) to IK, we obtain richer logics IT (adding axiom T), IK4 (adding axiom 4), and IS4 (adding both T and 4). Type systems with necessity modalities based on IK and IS4 have found applications in partial evaluation and staged computation (Davies and Pfenning 1996; Davies and Pfenning 2001), information-flow control (Miyamoto and Igarashi 2004), and recovering purity in an effectful language (Choudhury and Krishnaswami 2020). While type systems based on IT and IK4 do not seem to have any prior known programming applications, they are nevertheless interesting as objects of study that extend IK towards IS4.

Fitch-style modal lambda calculi (Borghuis 1994; Clouston 2018; Martini and Masini 1996) feature necessity modalities in a typed lambda calculus by extending the typing context with a delimiting "lock" operator (denoted by 🔒). In this paper, we consider the family of Fitch-style modal lambda calculi that correspond to the logics IK, IT, IK4, and IS4. These calculi extend the simply-typed lambda calculus (STLC) with a type constructor $\Box$, along with introduction and elimination rules for $\Box$ types formulated using the 🔒 operator. For instance, the calculus λ_{IK}, which corresponds to the logic IK, extends STLC with Rules $\Box$-Intro and $\lambda_{\mathrm{IK}}/\Box$-Elim, as summarized in Figure E.1. The rules for λ-abstraction and function application are formulated in the usual way—but note the modified variable rule Var!

The equivalence of terms in STLC is extended by Fitch-style calculi with the following rules for $\Box$ types, where the former states the β- (or computational) equivalence, and the latter states a type-directed η- (or extensional) equivalence.

$$\Box\text{-}\beta \qquad \mathsf{unbox}\,(\mathsf{box}\,t) \sim t$$

$$\Box\text{-}\eta \qquad \frac{\Gamma \vdash t : \Box A}{t \sim \mathsf{box}\,(\mathsf{unbox}\,t)}$$

We are interested in the problem of normalizing terms with respect to these equivalences. Traditionally, terms in a calculus are normalized by rewriting

$$\text{Ty} \quad A ::= \ldots \mid \Box A \qquad \text{Ctx} \quad \Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, \locked$$

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A}\ \text{Var}\ \locked \notin \Gamma' \qquad \frac{\Gamma, \locked \vdash t : A}{\Gamma \vdash \mathsf{box}\, t : \Box A}\ \Box\text{-Intro}$$

$$\frac{\Gamma \vdash t : \Box A}{\Gamma, \locked, \Gamma' \vdash \mathsf{unbox}_{\lambda_{\mathrm{IK}}}\, t : A}\ \lambda_{\mathrm{IK}}/\Box\text{-Elim}\ \locked \notin \Gamma'$$

Figure E.1.: Typing rules for λ_{IK} (omitting λ-abstraction and application)

them using rewrite rules formulated from these equivalences, and a term is said to be in *normal form* when it cannot be rewritten further. For example, we may formulate a rewrite rule $\mathsf{unbox}\,(\mathsf{box}\, t) \to t$ by orienting the $\Box$-β equivalence from left to right. This naive approach to formulating a rewrite rule, however, is insufficient for the $\Box$-η rule since normalizing with a rewrite rule $t \to \mathsf{box}\,(\mathsf{unbox}\, t)$ (for $\Gamma \vdash t : \Box A$) does not terminate as it can be applied infinitely many times. It is presumably for this reason that existing normalization results (Clouston 2018) for some of these calculi only consider β-equivalence.

While it may be possible to carefully formulate a more complex set of rewrite rules that take the context of application into consideration to guarantee termination (as done, for example, by Jay and Ghani (1995) for function and product types), the situation is further complicated for Fitch-style calculi by the fact that we must repeat such syntactic rewriting arguments separately for each calculus under consideration. The calculi λ_{IT}, λ_{IK4}, and λ_{IS4} differ from λ_{IK} only in the $\Box$-elimination rule, as summarized in Figure E.2. In spite of having identical syntax and term equivalences, each calculus demands different, tedious and seemingly ad hoc syntactic renaming lemmas (Clouston 2018, Lemmas 4.1 and 5.1) to prove normalization.

In this paper, we take a semantic approach to normalization, called normalization by evaluation (NbE) (Berger and Schwichtenberg 1991). NbE bypasses rewriting entirely, and instead normalizes terms by evaluating them in a suitable semantic model and then reifying values in the model as normal forms. For Fitch-style calculi, NbE can be developed by leveraging their possible-world semantics. To this end, we identify the parameters of the possible-world semantics for the calculi under consideration, and show that NbE models can be constructed by instantiating those parameters. The NbE approach exploits

$$\frac{\Gamma \vdash t : \Box A}{\Gamma, \Gamma' \vdash \mathsf{unbox}_{\lambda_{\mathrm{IT}}}\, t : A}\ \lambda_{\mathrm{IT}}/\Box\text{-Elim}\ \#_{\blacksquare}(\Gamma') \leqslant 1 \qquad \frac{\Gamma \vdash t : \Box A}{\Gamma, \blacksquare, \Gamma' \vdash \mathsf{unbox}_{\lambda_{\mathrm{IK4}}}\, t : A}\ \lambda_{\mathrm{IK4}}/\Box\text{-Elim}$$

$$\frac{\Gamma \vdash t : \Box A}{\Gamma, \Gamma' \vdash \mathsf{unbox}_{\lambda_{\mathrm{IS4}}}\, t : A}\ \lambda_{\mathrm{IS4}}/\Box\text{-Elim}$$

Figure E.2.: $\Box$-elimination rules for λ_{IT}, λ_{IK4}, and λ_{IS4}

the semantic overlap of the Fitch-style calculi in the possible-world semantics and isolates their differences to a specific parameter that determines the modal fragment, thus enabling the reuse of the evaluation machinery and many lemmas proved in the process.

In Section E.2, we begin by providing a brief overview of the main idea underlying this paper. We discuss the uniform interpretation of types for four Fitch-style calculi (λ_{IK}, λ_{IT}, λ_{IK4} and λ_{IS4}) in possible-world models and outline how NbE models can be constructed as instances. The *reification* mechanism that enables NbE is performed alike for all four calculi. In Section E.3, we construct an NbE model for λ_{IK} that yields a correct normalization algorithm, and then show how NbE models can also be constructed for λ_{IS4}, and for λ_{IT} and λ_{IK4} by slightly varying the instantiation. The calculi λ_{IK} and λ_{IS4} and their normalization algorithms have been implemented and verified correct (Valliappan, Ruch, and Tomé Cortiñas 2022a) in the proof assistant Agda (Abel, Allais, et al. 2005–2021).

NbE models and proofs of normalization in general have several useful consequences for term calculi. In Section E.4, we show how NbE models and the accompanying normalization algorithm can be used to prove meta-theoretic properties of Fitch-style calculi including completeness, decidability, and some standard results in modal logic in a *constructive* manner. In Section E.5, we discuss applications of our development to specific interpretations of the necessity modality in programming languages, and show (but do not mechanize) how application-specific properties that typically require semantic intervention can be proved syntactically. We show that properties similar to capability safety, noninterference, and binding-time correctness can be proved syntactically using normal forms of terms.

E.2. Main Idea

The main idea underlying this paper is that normalization can be achieved in a modular fashion for Fitch-style calculi by constructing NbE models as instances of their possible-world semantics. In this section, we observe that Fitch-style calculi can be interpreted in the possible-world semantics for intuitionistic modal logic with a minor refinement that accommodates the 🔒 operator, and give a brief overview of how we construct NbE models as instances.

Possible-World Semantics The possible-world semantics for intuitionistic modal logic (Božić and Došen 1984) is parameterized by a *frame* F and a *valuation* V_ι. A frame F is a triple (W, R_i, R_m) that consists of a type W of *worlds* along with two binary *accessibility* relations R_i (for "intuitionistic") and R_m (for "modal") on worlds that are required to satisfy certain conditions. An element $w : W$ can be thought of as a representation of the "knowledge state" about some "possible world" at a certain point in time. Then, $w\ R_i\ w'$ represents an increase in knowledge from w to w', and $w\ R_m\ v$ represents a possible passage from w to v. A valuation V_ι, on the other hand, is a family of types $V_{\iota,w}$ indexed by $w : W$ along with functions $\mathrm{wk}_{\iota,w,w'} : V_{\iota,w} \to V_{\iota,w'}$ whenever $w\ R_i\ w'$. An element $p : V_{\iota,w}$ can be thought of as "evidence" for (the knowledge of) the truth of the *atomic* proposition ι at the world w. The requirement for functions $\mathrm{wk}_{\iota,w,w'}$ enforces that the knowledge of the truth of ι at w is preserved as time moves on to w', and is neither forgotten nor contradicted by any new evidence learned at w'. There are no such requirements on a valuation V_ι with respect to the modal accessibility relation R_m.

Given a frame (W, R_i, R_m) and a valuation V_ι, we interpret (object) types A in *any* Fitch-style calculus as families of (meta) types $[\![A]\!]_w$ indexed by worlds $w : W$, following the work by Fischer-Servi (1981), Ewald (1986), Plotkin and Stirling (1986), and Simpson (1994) as below:

$$
\begin{aligned}
[\![\iota \quad\ ]\!]_w &= V_{\iota,w} \\
[\![A \Rightarrow B]\!]_w &= \forall w'.\, w\ R_i\ w' \to [\![A]\!]_{w'} \to [\![B]\!]_{w'} \\
[\![\Box A \quad]\!]_w &= \forall w'.\, w\ R_i\ w' \to \forall v.\, w'\ R_m\ v \to [\![A]\!]_v
\end{aligned}
$$

The nonmodal type formers are interpreted as in the Kripke semantics for intuitionistic propositional logic: the base type ι is interpreted using the valuation V_ι, and function types $A \Rightarrow B$ at $w : W$ are interpreted as *families* of functions $[\![A]\!]_{w'} \to [\![B]\!]_{w'}$ indexed by $w' : W$ such that $w\ R_i\ w'$. Recall that the generalization to families is necessary for the interpretation of function types to be sound.

As for the interpretation of modal types, at $w : W$ the types $\Box A$ are interpreted by families of elements $[\![A]\!]_v$ indexed by those $v : W$ that are accessible from w via some $w' : W$ such that $w\ R_i\ w'$ and $w'\ R_m\ v$. In other words, $\Box A$ is true at a world w if A is necessarily true in "the future", whichever concrete possibility this may turn out to be. We remark that the interpretation of $\Box A$ as $\forall v.\, w\ R_m\ v \to [\![A]\!]_v$, as in classical modal logic without the first quantifier $\forall w'.\, w\ R_i\ w'$, requires additional conditions (Božić and Došen 1984; Simpson 1994) on frames that (some of) the NbE models we construct do not satisfy.

In order to extend the possible-world semantics of intuitionistic modal logic to Fitch-style calculi, we must also provide an interpretation of contexts and the 🔒 operator, which is unique to the Fitch style, in particular:

$$\begin{aligned}
[\![\cdot \quad]\!]_w &= \top \\
[\![\Gamma, A]\!]_w &= [\![\Gamma]\!]_w \times [\![A]\!]_w \\
[\![\Gamma, 🔒]\!]_w &= \textstyle\sum_u [\![\Gamma]\!]_u \times u\ R_m\ w
\end{aligned}$$

The empty context $\cdot$ and the context extension Γ, A of a context Γ with a type A are interpreted as in the Kripke semantics for STLC by the terminal family and the Cartesian product of the families $[\![\Gamma]\!]$ and $[\![A]\!]$, respectively. While the interpretation of types $\Box A$ can be understood as a statement about the future, the interpretation of contexts $\Gamma, 🔒$ can be understood as a dual statement about the past: $\Gamma, 🔒$ is true at a world w if Γ is true at *some* world u for which w is a possibility, i.e. $u\ R_m\ w$.

With the interpretation of contexts Γ and types A as (W, R_i)-indexed families $[\![\Gamma]\!]$ and $[\![A]\!]$ at hand, the interpretation of terms $t : \Gamma \vdash A$, also known as *evaluation*, in a possible-world model is given by a function $[\![-]\!] : \Gamma \vdash A \to (\forall w.\, [\![\Gamma]\!]_w \to [\![A]\!]_w)$ as follows. Clouston (2018) shows that the interpretation of STLC in Cartesian closed categories (CCCs) extends to an interpretation of Fitch-style calculi in any CCCequipped with an adjunction by interpreting $\Box$ and 🔒 by the right and left adjoint as well as box and unbox using the right and left adjuncts, respectively. The key idea here is that, correspondingly, the interpretation of terms in the nonmodal fragment of Fitch-style calculi using the familiar CCCstructure on (W, R_i)-indexed families extends to the modal fragment: the interpretation of $\Box$ in a possible-world model has a left adjoint that is denoted by our interpretation of 🔒. In summary, the possible-world interpretation of Fitch-style calculi can be given by instantiation of Clouston's *generic* interpretation in CCCsequipped with an adjunction.

Constructing NbE Models as Instances To construct an NbE model for Fitch-style calculi, we must construct a possible-world model with a function quote :

$(\forall w.\, [\![\Gamma]\!]_w \to [\![A]\!]_w) \to \Gamma \vdash_{\text{NF}} A$ that inverts the denotation $(\forall w.\, [\![\Gamma]\!]_w \to [\![A]\!]_w)$ of a term to a derivation $\Gamma \vdash_{\text{NF}} A$ in normal form. The normal forms for the modal fragment of λ_{IK} are defined below, where $\Gamma \vdash_{\text{NE}} A$ denotes a special case of normal forms known as *neutral elements*.

$$\frac{\Gamma, \blacksquare \vdash_{\text{NF}} t : A}{\Gamma \vdash_{\text{NF}} \mathsf{box}\, t : \Box A}\ \text{NF/}\Box\text{-Intro} \qquad \frac{\Gamma \vdash_{\text{NE}} t : \Box A}{\Gamma, \blacksquare, \Gamma' \vdash_{\text{NE}} \mathsf{unbox}_{\lambda_{\text{IK}}}\, t : A}\ \blacksquare \notin \Gamma'\ \lambda_{\text{IK}}/\text{NE}/\Box\text{-Elim}$$

The normal forms for λ_{IT}, λ_{IK4}, and λ_{IS4} are defined similarly by varying the elimination rule as in their term typing rules in Figure E.2.

Following the work on NbE for STLC with possible-world[1] models (Coquand 2002), we instantiate the parameters that define possible-world models for Fitch-style calculi as follows: we pick contexts for W, *order-preserving embeddings* (sometimes called "weakenings", defined in the next section) $\Gamma \leqslant \Gamma'$ for $\Gamma\ R_i\ \Gamma'$, and neutral derivations $\Gamma \vdash_{\text{NE}} \iota$ as the valuation $V_{\iota,\Gamma}$. It remains for us to instantiate the parameter R_m and show that this model supports the quote function.

The instantiation of the modal parameter R_m in the possible-world semantics varies for each calculus and captures the difference between them. Recall that the syntaxes of the four calculi only differ in their elimination rules for $\Box$ types. When viewed through the lens of the possible-world semantics, this difference can be generalized as follows:

$$\frac{\Delta \vdash t : \Box A}{\Gamma \vdash \mathsf{unbox}\, t : A}\ (\Delta \lhd \Gamma)\ \Box\text{-Elim}$$

We generalize the relationship between the context in the premise and the context in the conclusion using a generic modal accessibility relation $\lhd$ between contexts. When viewed as a candidate for instantiating the R_m relation, this rule states that if $\Box A$ is derivable in some past world Δ, then we may derive A in the current world Γ. The various $\Box$-elimination rules for Fitch-style calculi can be viewed as instances of this generalized rule, where we define $\lhd$ in accordance with $\Box$-elimination rule of the calculus under consideration. For example, for λ_{IK}, we observe that the context of the premise in Rule $\lambda_{\text{IK}}/\Box$-Elim is Γ and that of the conclusion is $\Gamma, \blacksquare, \Gamma'$ such that $\blacksquare \notin \Gamma'$, and thus define $\Delta \lhd_{\lambda_{\text{IK}}} \Gamma$ as $\exists \Delta'.\, \blacksquare \notin \Delta' \wedge \Gamma = \Delta, \blacksquare, \Delta'$. Similarly, we define $\Delta \lhd_{\lambda_{\text{IS4}}} \Gamma$ as $\exists \Delta'.\, \Gamma = \Delta, \Delta'$ for λ_{IS4}, and follow this recipe for λ_{IT} and λ_{IK4}. Accordingly, we instantiate the

[1] also called "Kripke" or "Kripke-style"

R_m parameter in the NbE model with the corresponding definition of $\lhd$ in the calculus under consideration.

A key component of implementing the quote function in NbE models is *reification*, which is implemented by a family of functions $\mathrm{reify}_A : \forall \Gamma.\, [\![A]\!]_\Gamma \to \Gamma \vdash_{\mathrm{NF}} A$ indexed by a type A. While its implementation for the simply-typed fragment follows the standard, for the modal fragment we are required to give an implementation of $\mathrm{reify}_{\Box A} : \forall \Gamma.\, [\![\Box A]\!]_\Gamma \to \Gamma \vdash_{\mathrm{NF}} \Box A$. To reify a value of $[\![\Box A]\!]_\Gamma$, we first observe that $[\![\Box A]\!]_\Gamma = \forall \Gamma'.\, \Gamma \leqslant \Gamma' \to \forall \Delta.\, \Gamma' \lhd \Delta \to [\![A]\!]_\Delta$ by definition of $[\![-]\!]$ and the instantiations of R_i with $\leqslant$ and R_m with $\lhd$. By picking Γ for Γ' and $\Gamma, 🔒$ for Δ, we get $[\![A]\!]_{\Gamma,🔒}$ since $\leqslant$ is reflexive and it can be shown that $\Gamma \lhd \Gamma, 🔒$ holds for the calculi under consideration. By reifying the value $[\![A]\!]_{\Gamma,🔒}$ recursively, we get a normal form $\Gamma, 🔒 \vdash_{\mathrm{NF}} n : A$, which can be used to construct the desired normal form $\Gamma \vdash_{\mathrm{NF}} \mathsf{box}\, n : \Box A$ using the rule NF/$\Box$-Intro.

E.3. Possible-World Semantics and NbE

In this section, we elaborate on the previous section by defining possible-world models and showing that Fitch-style calculi can be interpreted soundly in these models. Following this, we outline the details of constructing NbE models as instances. We begin with the calculus λ_{IK}, and then show how the same results can be achieved for the other calculi.

Before discussing a concrete calculus, we present some of their commonalities.

Types, Contexts and Order-Preserving Embeddings The grammar of types and typing contexts for Fitch-style is the following.

$$\text{Ty} \quad A ::= \iota \mid A \Rightarrow B \mid \Box A \qquad\qquad \text{Ctx} \quad \Gamma ::= \cdot \mid \Gamma, A \mid \Gamma, 🔒$$

Types are generated by an uninterpreted base type ι, function types $A \Rightarrow B$, and modal types $\Box A$, and typing contexts are "snoc" lists of types and locks.

We define the relation of *order-preserving embeddings* (OPE) on typing contexts in Figure E.3. An OPE $\Gamma \leqslant \Gamma'$ embeds the context Γ into another context Γ' while preserving the order of types and the order and number of locks in Γ.

$$\mathsf{base} : \cdot \leqslant \cdot \qquad \frac{o : \Gamma \leqslant \Gamma'}{\mathsf{drop}\, o : \Gamma \leqslant \Gamma', A} \qquad \frac{o : \Gamma \leqslant \Gamma'}{\mathsf{keep}\, o : \Gamma, A \leqslant \Gamma', A}$$

$$\frac{o : \Gamma \leqslant \Gamma'}{\mathsf{keep}_{\color{black}\blacksquare}\, o : \Gamma, \color{black}\blacksquare \leqslant \Gamma', \color{black}\blacksquare}$$

Figure E.3.: Order-preserving embeddings

E.3.1. The Calculus $\lambda_{\mathbf{IK}}$

Terms, Substitutions and Equational Theory

To define the intrinsically-typed syntax and equational theory of λ_{IK}, we first define a modal accessibility relation on contexts $\Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma$, which expresses that context Γ extends $\Delta, \color{black}\blacksquare$ to the right without adding locks. Note that $\Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma$ exactly when $\exists \Delta'. \color{black}\blacksquare \notin \Delta' \wedge \Gamma = \Delta, \color{black}\blacksquare, \Delta'$.

$$\mathsf{nil} : \Gamma \lhd_{\lambda_{\mathrm{IK}}} \Gamma, \color{black}\blacksquare \qquad \frac{e : \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma}{\mathsf{var}\, e : \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma, A}$$

Figure E.4.: Modal accessibility relation on contexts (λ_{IK})

Figure E.5 presents the intrinsically-typed syntax of λ_{IK}. We will use both $\Gamma \vdash t : A$ and $t : \Gamma \vdash A$ to say that t denotes an (intrinsically-typed) term of type A in context Γ, and similarly for substitutions, which will be defined below. Instead of named variables as in Figure E.1, variables are defined using De Bruijn indices in a separate judgement $\Gamma \vdash_{\mathrm{VAR}} A$. The introduction and elimination rules for function types are like those in STLC, and the introduction rule for the type $\Box A$ is similar to that of Figure E.1. The elimination rule $\lambda_{\mathrm{IK}}/\Box$-Elim is defined using the modal accessibility relation $\Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma$ which relates the contexts in the premise and the conclusion, respectively. This relation replaces the side condition ($\color{black}\blacksquare \notin \Gamma'$) in Figure E.1 and other $\Box$-elimination rules in Sections E.1 and E.2. Note that formulating the rule for the term $\mathsf{unbox}_{\lambda_{\mathrm{IK}}}$ with $e : \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma$ as a second premise is in sharp contrast to Clouston (2018, Fig. 1) where the relation is not mentioned in the term but formulated as the *side condition* $\Gamma = \Delta, \color{black}\blacksquare, \Gamma'$ for some lock-free Γ'.

A term $\Gamma \vdash t : A$ can be *weakened*, which is a special case of *renaming*, with an

$$\textsc{Var-Zero}\quad \Gamma, A \vdash_{\text{VAR}} \mathsf{zero} : A$$

$$\textsc{Var-Succ}\quad \frac{\Gamma \vdash_{\text{VAR}} v : A}{\Gamma, B \vdash_{\text{VAR}} \mathsf{succ}\, v : A}$$

$$\textsc{Var}\quad \frac{\Gamma \vdash_{\text{VAR}} v : A}{\Gamma \vdash \mathsf{var}\, v : A}$$

$$\Rightarrow\textsc{-Intro}\quad \frac{\Gamma, A \vdash t : B}{\Gamma \vdash \lambda\, t : A \Rightarrow B}$$

$$\Rightarrow\textsc{-Elim}\quad \frac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{app}\, t\, u : B}$$

$$\Box\textsc{-Intro}\quad \frac{\Gamma, \blacksquare \vdash t : A}{\Gamma \vdash \mathsf{box}\, t : \Box A}$$

$$\lambda_{\text{IK}}/\Box\textsc{-Elim}\quad \frac{\Delta \vdash t : \Box A \qquad e : \Delta \lhd_{\lambda_{\text{IK}}} \Gamma}{\Gamma \vdash \mathsf{unbox}_{\lambda_{\text{IK}}}\, t\, e : A}$$

Figure E.5.: Intrinsically-typed terms of λ_{IK}

OPE (see Figure E.3) using a function $\text{wk} : \Gamma \leqslant \Gamma' \to \Gamma \vdash A \to \Gamma' \vdash A$. Given an OPE $o : \Gamma \leqslant \Gamma'$, renaming the term using wk yields a term $\Gamma' \vdash \text{wk}\, o\, t : A$ in the weaker context Γ'. The unit element for wk is the identity OPE $\mathsf{id}_{\leqslant} : \Gamma \leqslant \Gamma$, i.e. $\text{wk}\, \mathsf{id}_{\leqslant}\, t = t$. Renaming arises naturally when evaluating terms and in specifying the equational theory (e.g. in the η rule of function type).

$$\Gamma \vdash_{\text{S}} \mathsf{empty} : \cdot \qquad \frac{\Gamma \vdash_{\text{S}} s : \Delta \qquad \Gamma \vdash t : A}{\Gamma \vdash_{\text{S}} \mathsf{ext}\, s\, t : \Delta, A} \qquad \frac{\Theta \vdash_{\text{S}} s : \Delta \qquad e : \Theta \lhd_{\lambda_{\text{IK}}} \Gamma}{\Gamma \vdash_{\text{S}} \mathsf{ext}_{\blacksquare}\, s\, e : \Delta, \blacksquare}$$

Figure E.6.: Substitutions for λ_{IK}

Substitutions for λ_{IK} are inductively defined in Figure E.6. A judgement $\Gamma \vdash_{\text{S}} s : \Delta$ denotes a substitution for a context Δ in the context Γ. Applying a substitution to a term $\Delta \vdash t : A$, i.e. $\text{subst}\, s\, t : \Gamma \vdash A$, yields a term in the context Γ. The substitution $\mathsf{id}_{\text{S}} : \Gamma \vdash_{\text{S}} \Gamma$ denotes the identity substitution, which exists for all Γ. As usual, it can be shown that terms are closed under the application of a substitution, and that it preserves the identity, i.e. $\text{subst}\, \mathsf{id}_{\text{S}}\, t = t$. Substitutions are also closed under renaming and this operation preserves the identity as well.

The equational theory for λ_{IK}, omitting congruence rules, is specified in Figure E.7. As discussed earlier, λ_{IK} extends the usual rules in STLC (Rules $\Rightarrow$-β and $\Rightarrow$-η) with rules for the $\Box$ type (Rules $\Box$-β and $\Box$-η). The function factor : $\Delta \lhd_{\lambda_{\text{IK}}} \Gamma \to \Delta, \blacksquare \leqslant \Gamma$, in Rule $\Box$-β, maps an element of the modal accessibility

relation $e : \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma$ to an OPE $\Delta, \mathbin{🔒} \leqslant \Gamma$. This is possible because the context Γ does not have any lock to the right of $\Delta, \mathbin{🔒}$.

$$\Rightarrow\text{-}\beta\ \frac{\Gamma, A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{app}\,(\lambda\, t)\, u \sim \mathrm{subst}\,(\mathsf{ext}\,\mathsf{id}_{\mathrm{s}}\, u)\, t}$$

$$\Rightarrow\text{-}\eta\ \frac{\Gamma \vdash t : A \Rightarrow B}{\Gamma \vdash t \sim \lambda\,(\mathsf{app}\,(\mathrm{wk}\,(\mathsf{drop}\,\mathsf{id}_{\leqslant})\, t)\,(\mathsf{var}\,\mathsf{zero}))}$$

$$\Box\text{-}\beta\ \frac{\Delta, \mathbin{🔒} \vdash t : A \qquad e : \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma}{\Gamma \vdash \mathsf{unbox}_{\lambda_{\mathrm{IK}}}\,(\mathsf{box}\, t)\, e \sim \mathrm{wk}\,(\mathrm{factor}\, e)\, t} \qquad \Box\text{-}\eta\ \frac{\Gamma \vdash t : \Box A}{\Gamma \vdash t \sim \mathsf{box}\,(\mathsf{unbox}_{\lambda_{\mathrm{IK}}}\, t\, \mathsf{nil})}$$

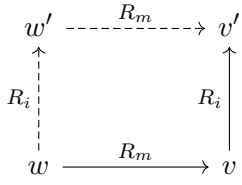
Figure E.7.: Equational theory for λ_{IK}

Possible-World Semantics

A possible-world model is defined using the notion of a possible-world frame as below. We work in a constructive type-theoretic metalanguage, and denote the universe of types in this language by Type.

Definition E.3.1 (Possible-world frame)**.** A frame F is given by a triple (W, R_i, R_m) consisting of a type W : Type and two relations R_i and R_m: $W \times W \to$ Type on W such that the following conditions are satisfied:

- R_i is *reflexive* and *transitive*
- if $w\ R_m\ v$ and $v\ R_i\ v'$ then there exists some $w' : W$ such that $w\ R_i\ w'$ and $w'\ R_m\ v'$; this *factorization* condition can be pictured as an implication $R_m\,;R_i \subseteq R_i\,;R_m$ or diagrammatically as follows:



(note that neither w' nor the proofs of relatedness are required to be unique, nor will they all be in the frames that we will consider)

Definition E.3.2 (Possible-world model)**.** A possible-world model $\mathcal{M}$ is given by a tuple (F, V) consisting of a frame F (see Definition E.3.1) and a W-indexed family $V_\iota : W \to \text{Type}$ (called the *valuation* of the base type) such that $\forall w, w'.\, w\ R_i\ w' \to V_{\iota,w} \to V_{\iota,w'}$.

We have omitted coherence conditions from these Definitions for readability. Those conditions stem from the proof relevance of the relations and predicates involved. They will be satisfied by the models we will construct, and will also be given below for completeness.

The types and typing contexts in λ_{IK} are interpreted in a possible-world model via the interpretation functions $[\![-]\!]$ defined in Section E.2. To evaluate terms, we must first prove the following *monotonicity* lemma. This lemma is well-known as a requirement to give a sound interpretation of the function type in an arbitrary possible-world model, and can be thought of as the semantic generalization of renaming in terms.

Lemma E.3.1 (Monotonicity)**.** *In every possible-world model $\mathcal{M}$, for every type A and worlds w and w', we have a function* $\text{wk}_A : w\ R_i\ w' \to [\![A]\!]_w \to [\![A]\!]_{w'}$*. And similarly, for every context Γ, a function* $\text{wk}_\Gamma : w\ R_i\ w' \to [\![\Gamma]\!]_w \to [\![\Gamma]\!]_{w'}$*.*

We evaluate terms in λ_{IK} in a possible-world model as follows.

$$
\begin{aligned}
&[\![-]\!] : \Gamma \vdash A \to (\forall w.\, [\![\Gamma]\!]_w \to [\![A]\!]_w)\\
&[\![\mathsf{var}\, v \qquad\quad]\!]\, \gamma = \text{lookup}\, v\, \gamma\\
&[\![\lambda\, t \qquad\quad]\!]\, \gamma = \lambda i.\, \lambda a.\, [\![t]\!]\, (\text{wk}\, i\, \gamma, a)\\
&[\![\mathsf{app}\, t\, u \qquad]\!]\, \gamma = ([\![t]\!]\, \gamma)\, \mathsf{id}_{\leqslant}\, ([\![u]\!]\, \gamma)\\
&[\![\mathsf{box}\, t \qquad\quad]\!]\, \gamma = \lambda i.\, \lambda m.\, [\![t]\!]\, (\text{wk}\, i\, \gamma, m)\\
&[\![\mathsf{unbox}_{\lambda_{\text{IK}}}\, t\, e]\!]\, \gamma = [\![t]\!]\, \delta\, \mathsf{id}_{\leqslant}\, m\\
&\qquad \text{where } (\delta, m) = \text{trim}_{\lambda_{\text{IK}}}\, \gamma\, e
\end{aligned}
$$

The evaluation of terms in the simply-typed fragment is standard, and resembles the evaluator of STLC. Variables are interpreted by a lookup function that projects values from an environment, and λ-abstraction and application are evaluated using their semantic counterparts. To evaluate λ-abstraction, we must construct a semantic function $\forall w'.\, w\ R_i\ w' \to [\![A]\!]_{w'} \to [\![B]\!]_{w'}$ using the given term $\Gamma, A \vdash t : B$ and environment $\gamma : [\![\Gamma]\!]_w$. We achieve this by recursively evaluating t in an environment that extends γ appropriately using the semantic arguments $i : w\ R_i\ w'$ and $a : [\![A]\!]_{w'}$. We use the monotonicity lemma to "transport" $[\![\Gamma]\!]_w$ to $[\![\Gamma]\!]_{w'}$, and construct an environment of type $[\![\Gamma]\!]_{w'} \times [\![A]\!]_{w'}$ for recursively evaluating t, which produces the desired result of type $[\![B]\!]_{w'}$. Application is evaluated by simply recursively evaluating the

applied terms and applying them in the semantics with a value $\mathsf{id}_{\leqslant} : w\ R_i\ w$, which is available since R_i is reflexive.

In the modal fragment, to evaluate the term $\Gamma \vdash \mathsf{box}\, t : \Box A$ with $\gamma : [\![\Gamma]\!]_w$, we must construct a function of type $\forall w'.\, w\ R_i\ w' \to \forall v.\, w'\ R_m\ v \to [\![A]\!]_v$. Using the semantic arguments $i : w\ R_i\ w'$ and $m : w'\ R_m\ v$, we recursively evaluate the term $\Gamma, \blacksquare \vdash t : A$ in the extended environment $(\mathrm{wk}\, i\, \gamma, m) : [\![\Gamma, \blacksquare]\!]_v$, since $[\![\Gamma, \blacksquare]\!]_v = \sum_{w'} [\![\Gamma]\!]_{w'} \times w'\ R_m\ v$. On the other hand, the term $\Gamma \vdash \mathsf{unbox}_{\lambda_{\mathrm{IK}}}\, t\, e : A$ with $e : \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma$ and $\Delta \vdash t : \Box A$, for some Δ, must be evaluated with an environment $\gamma : [\![\Gamma]\!]_w$. To recursively evaluate the term $\Delta \vdash t : \Box A$, we must first discard the part of the environment γ that substitutes the types in the extension of $\Delta, \blacksquare$. This is achieved using the function $\mathrm{trim}_{\lambda_{\mathrm{IK}}} : [\![\Gamma]\!]_w \to \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma \to [\![\Delta, \blacksquare]\!]_w$ that projects γ to produce an environment $\delta : [\![\Delta]\!]_{v'}$ and a value $m : v'\ R_m\ w$. We evaluate t with δ and apply the resulting function of type $\forall v.\, v\ R_i\ v' \to \forall w.\, v'\ R_m\ w \to [\![A]\!]_w$ to $\mathsf{id}_{\leqslant}$ and m to return the desired result.

We state the soundness of λ_{IK} with respect to the possible-world semantics before we instantiate it with the NbE model that we will construct in the next section. We note that the soundness proof relies on the possible-world models to satisfy coherence conditions that we have omitted from Definitions E.3.1 and E.3.2 but that will be satisfied by the NbE models. Specifically, W and R_i together with the transitivity and reflexivity proofs $\mathrm{trans_i}$ and $\mathrm{refl_i}$ for R_i need to form a category $\mathscr{W}$, i.e. $\mathrm{trans_i}$ needs to be associative and $\mathrm{refl_i}$ needs to be a unit for $\mathrm{trans_i}$; the proofs of the factorization condition need to satisfy the functoriality laws $\mathrm{factor_i}\, m\, (\mathrm{refl_i}\, v) = \mathrm{refl_i}\, w$, $\mathrm{factor_m}\, m\, (\mathrm{refl_i}\, v) = m$, $\mathrm{factor_i}\, m\, (\mathrm{trans_i}\, i\, j) = \mathrm{trans_i}\, (\mathrm{factor_i}\, m\, i)\, (\mathrm{factor_i}\, m'\, j)$ and $\mathrm{factor_m}\, m\, (\mathrm{trans_i}\, i\, j) = \mathrm{factor_m}\, m'\, j$ where $m' \coloneqq \mathrm{factor_m}\, m\, i : w'\ R_m\ v'$ denotes the modal accessibility proof produced by the first factorization of $m : w\ R_m\ v$ and $i : v\ R_i\ v'$; and V_ι together with the monotonicity proof wk_ι needs to form a functor on the category $\mathscr{W}$, i.e. $\mathrm{wk}_\iota\, (\mathrm{refl_i}\, w)$ needs to be equal to the identity function on $V_{\iota,w}$ and $\mathrm{wk}_\iota\, (\mathrm{trans_i}\, i\, j)$ needs to be equal to the composite $\mathrm{wk}_\iota\, j \circ \mathrm{wk}_\iota\, i$.

Theorem E.3.1. *Let $\mathcal{M}$ be any possible-world model (see Definition E.3.2). If two terms t and $u : \Gamma \vdash A$ of λ_{IK} are equivalent (see Figure E.7) then the functions $[\![t]\!]$ and $[\![u]\!] : \forall w.\, [\![\Gamma]\!]_w \to [\![A]\!]_w$ as determined by $\mathcal{M}$ are equal.*

Proof. Let $\mathcal{M}$ be a possible-world model with underlying frame $F = (W, R_i, R_m)$. Denote the category whose objects are worlds $w : W$ and whose morphisms are proofs $i : w\ R_i\ w'$ by $\mathcal{C}$. The frame F can be seen as determining an adjunction $\blacksquare \dashv \Box$ on the category of presheaves indexed by the category $\mathcal{C}$, which is moreover well-known to be Cartesian closed. The interpretation $[\![-]\!]$

can then be seen as factoring through the categorical semantics described in Clouston (2018, Section 2.3), of which the category of presheaves over $\mathcal{C}$ is an instance by virtue of its Cartesian closure and equipment with an adjunction. We can therefore conclude by applying Clouston (2018, Theorem 2.8 (Categorical Soundness) and remark below that). □

NbE Model

The normal forms of terms in λ_{IK} are defined along with neutral elements in a mutually recursive fashion by the judgements $\Gamma \vdash_{\mathrm{NF}} A$ and $\Gamma \vdash_{\mathrm{NE}} A$, respectively, in Figure E.8. Intuitively, a normal form may be thought of as a value, and a neutral element may be thought of as a "stuck" computation. We extend the standard definition of normal forms and neutral elements in STLC with Rules NF/□-INTRO and λ_{IK}/NE/□-ELIM.

$$\textsc{Ne/Var}\ \frac{\Gamma \vdash_{\mathrm{VAR}} v : A}{\Gamma \vdash_{\mathrm{NE}} \mathsf{var}\, v : A} \qquad \textsc{Nf/Up}\ \frac{\Gamma \vdash_{\mathrm{NE}} n : \iota}{\Gamma \vdash_{\mathrm{NF}} \mathsf{up}\, n : \iota} \qquad \textsc{Nf/}\Rightarrow\textsc{-Intro}\ \frac{\Gamma, A \vdash_{\mathrm{NF}} n : B}{\Gamma \vdash_{\mathrm{NF}} \lambda\, n : A \Rightarrow B}$$

$$\textsc{Ne/}\Rightarrow\textsc{-Elim}\ \frac{\Gamma \vdash_{\mathrm{NE}} n : A \Rightarrow B \qquad \Gamma \vdash_{\mathrm{NF}} m : A}{\Gamma \vdash_{\mathrm{NE}} \mathsf{app}\, n\, m : B} \qquad \textsc{Nf/}\Box\textsc{-Intro}\ \frac{\Gamma, \blacksquare \vdash_{\mathrm{NF}} n : A}{\Gamma \vdash_{\mathrm{NF}} \mathsf{box}\, n : \Box A}$$

$$\lambda_{\mathrm{IK}}\textsc{/Ne/}\Box\textsc{-Elim}\ \frac{\Delta \vdash_{\mathrm{NE}} n : \Box A \qquad e : \Delta \lhd_{\lambda_{\mathrm{IK}}} \Gamma}{\Gamma \vdash_{\mathrm{NE}} \mathsf{unbox}_{\lambda_{\mathrm{IK}}}\, n\, e : A}$$

Figure E.8.: Normal forms and neutral elements in λ_{IK}

Recall that an NbE model for a given calculus C is a particular kind of model $\mathcal{M}$ that comes equipped with a function quote : $\mathcal{M}([\![\Gamma]\!], [\![A]\!]) \to \Gamma \vdash_{\mathrm{NF}} A$ satisfying $t \sim \mathrm{quote}\, [\![t]\!]$ for all terms $t : \Gamma \vdash A$ where $[\![-]\!]$ denotes the *generic* evaluation function for C.

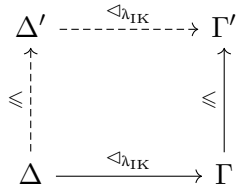
Using the relations defined in Figures E.3 and E.4, we construct an NbE model for λ_{IK} by instantiating the parameters that define a *possible-world* model as follows.

- Worlds as contexts: $W = \mathrm{Ctx}$
- Relation R_i as order-preserving embeddings: $\Gamma\ R_i\ \Gamma' = \Gamma \leqslant \Gamma'$

- Relation R_m as extensions of a "locked" context: $\Delta\ R_m\ \Gamma = \Delta \lhd_{\lambda_{\text{IK}}} \Gamma$
- Valuation V_ι as neutral elements: $V_{\iota,\Gamma} = \Gamma \vdash_{\text{NE}} \iota$

The condition that the valuation must satisfy $\text{wk}_A : \Gamma \leqslant \Gamma' \to \Gamma \vdash_{\text{NE}} A \to \Gamma' \vdash_{\text{NE}} A$, for all types A, can be shown by induction on the neutral term $\Gamma \vdash_{\text{NE}} A$. To show that this model is indeed a possible-world model, it remains for us to show that the frame conditions are satisfied.

The first frame condition states that OPEs must be reflexive and transitive, which can be shown by structural induction on the context and definition of OPEs, respectively. The second frame condition states that given $\Delta \lhd_{\lambda_{\text{IK}}} \Gamma$ and $\Gamma \leqslant \Gamma'$ there is a $\Delta' : \text{Ctx}$ such that $\Delta \leqslant \Delta'$ and $\Delta' \lhd_{\lambda_{\text{IK}}} \Gamma'$,



which can be shown by constructing a function by simultaneous recursion on OPEs and the modal accessibility relation.

Observe that the instantiation of the monotonicity lemma in the NbE model states that we have the functions $\text{wk}_A : \Gamma \leqslant \Gamma' \to [\![A]\!]_\Gamma \to [\![A]\!]_{\Gamma'}$ and $\text{wk}_\Delta : \Gamma \leqslant \Gamma' \to [\![\Delta]\!]_\Gamma \to [\![\Delta]\!]_{\Gamma'}$, which allow denotations of types and contexts to be renamed with respect to an OPE.

To implement the function quote, we first implement *reification* and *reflection*, using two functions $\text{reify}_A : [\![A]\!]_\Gamma \to \Gamma \vdash_{\text{NF}} A$ and $\text{reflect}_A : \Gamma \vdash_{\text{NE}} A \to [\![A]\!]_\Gamma$, respectively. Reification converts a semantic value to a normal form, while reflection converts a neutral element to a semantic value. They are implemented as follows by induction on the index type A.

$$
\begin{aligned}
&\text{reify}_{A,\Gamma} : [\![A]\!]_\Gamma \to \Gamma \vdash_{\text{NF}} A \\
&\text{reify}_{\iota,\Gamma}\ \ n = \mathsf{up}\, n \\
&\text{reify}_{A\Rightarrow B,\Gamma}\ f = \lambda\, (\text{reify}_{B,(\Gamma,A)}(f\ (\mathsf{drop}\, \mathsf{id}_{\leqslant})\, \text{fresh}_{A,\Gamma})) \\
&\text{reify}_{\Box A,\Gamma}\ \ b = \mathsf{box}\, (\text{reify}_{A,(\Gamma,🔒)}(b\, \mathsf{id}_{\leqslant}\, \mathsf{nil}))
\end{aligned}
$$

$$
\begin{aligned}
&\text{reflect}_{A,\Gamma} : \Gamma \vdash_{\text{NE}} A \to [\![A]\!]_\Gamma \\
&\text{reflect}_{\iota,\Gamma}\ \ n = n \\
&\text{reflect}_{A\Rightarrow B,\Gamma}\ n = \lambda(o : \Gamma \leqslant \Gamma').\, \lambda a.\, \text{reflect}_{B,\Gamma}(\mathsf{app}\, (\text{wk}_{A\Rightarrow B}\, o\, n)\, (\text{reify}_{A,\Gamma'}\, a)) \\
&\text{reflect}_{\Box A,\Gamma}\ \ n = \lambda(o : \Gamma \leqslant \Gamma').\, \lambda(e : \Gamma' \lhd_{\lambda_{\text{IK}}} \Delta).\, \text{reflect}_{A,\Delta}(\mathsf{unbox}_{\lambda_{\text{IK}}}\, (\text{wk}_{\Box A}\, o\, n)\, e)
\end{aligned}
$$

For the function type, we recursively reify the body of the λ-abstraction by applying the given semantic function f with suitable arguments, which are an OPE $\mathsf{drop}\,\mathsf{id}_{\leqslant} : \Gamma \leqslant \Gamma, A$ and a value $\mathrm{fresh}_{A,\Gamma} = \mathrm{reflect}_{A,(\Gamma,A)}\,(\mathsf{var}\,\mathsf{zero}) : [\![A]\!]_{\Gamma,A}$, which is the De Bruijn index equivalent of a fresh variable. Reflection, on the other hand, recursively reflects the application of a neutral $\Gamma \vdash_{\mathrm{NE}} n : A \Rightarrow B$ to the reification of the semantic argument $a : [\![A]\!]_{\Gamma'}$ for an OPE $o : \Gamma \leqslant \Gamma'$. Similarly, for the $\Box$ type, we recursively reify the body of box by applying the given semantic function $b : \forall\Gamma.\,\Gamma \leqslant \Gamma' \to \forall\Delta.\,\Gamma' \lhd_{\lambda_{\mathrm{IK}}} \Delta \to [\![A]\!]_{\Delta}$ to suitable arguments $\mathsf{id}_{\leqslant} : \Gamma \leqslant \Gamma$ and the empty context extension $\mathsf{nil} : \Gamma \lhd_{\lambda_{\mathrm{IK}}} \Gamma, \text{🔒}$. Reflection also follows a similar pursuit by reflecting the application of the neutral $\Gamma \vdash_{\mathrm{NE}} n : \Box A$ to the eliminator unbox.

Equipped with reification, we implement quote (as seen below), by applying the given denotation of a term, a function $f : \forall\Delta.\,[\![\Gamma]\!]_{\Delta} \to [\![A]\!]_{\Delta}$, to the identity environment $\mathrm{freshEnv}_{\Gamma} : [\![\Gamma]\!]_{\Gamma}$, and then reifying the resulting value. The construction of the value $\mathrm{freshEnv}_{\Gamma}$ is the De Bruijn index equivalent of generating an environment with fresh variables.

$$
\begin{aligned}
&\mathrm{quote} : (\forall\Delta.\,[\![\Gamma]\!]_{\Delta} \to [\![A]\!]_{\Delta}) \to \Gamma \vdash_{\mathrm{NF}} A\\
&\mathrm{quote}\, f = \mathrm{reify}_{A,\Gamma}\,(f\,\mathrm{freshEnv}_{\Gamma})
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{freshEnv}_{\Gamma} : [\![\Gamma]\!]_{\Gamma}\\
&\mathrm{freshEnv}_{\cdot} \quad = ()\\
&\mathrm{freshEnv}_{\Gamma,A} = (\mathrm{wk}\,(\mathsf{drop}\,\mathsf{id}_{\leqslant})\,\mathrm{freshEnv}_{\Gamma}, \mathrm{fresh}_{A,\Gamma})\\
&\mathrm{freshEnv}_{\Gamma,\text{🔒}} = (\mathrm{freshEnv}_{\Gamma}, \mathsf{nil}\,)
\end{aligned}
$$

To prove that the function quote is indeed a retraction of evaluation, we follow the usual logical relations approach. As seen in Figure E.9, we define a relation L_A indexed by a type A that relates a term $\Gamma \vdash t : A$ to its denotation $a : [\![A]\!]_{\Gamma}$ as $\mathrm{L}_A\,t\,a$. From a proof of $\mathrm{L}_A\,t\,a$, it can be shown that $t \sim \mathrm{reify}_A\,a$. This relation is extended to contexts as L_{Δ}, for some context Δ, which relates a substitution $\Gamma \vdash s : \Delta$ to its denotation $\delta : [\![\Delta]\!]_{\Gamma}$ as $\mathrm{L}_{\Delta}\,s\,\delta$.

For the logical relations, we then prove the so-called fundamental theorem.

Proposition E.3.1 (Fundamental theorem)**.** *Given a term $\Delta \vdash t : A$, a substitution $\Gamma \vdash_s s : \Delta$ and a value $\delta : [\![\Delta]\!]_{\Gamma}$, if $\mathrm{L}_{\Delta,\Gamma}\,s\,\delta$ then $\mathrm{L}_{A,\Gamma}\,(\mathrm{subst}\,s\,t)\,([\![t]\!]\,\delta)$.*

We conclude this section by stating the normalization theorem for λ_{IK}.

Proposition E.3.1 entails that $\mathrm{L}_{A,\Delta}\,(\mathrm{subst}\,\mathsf{id}_{\mathrm{S}}\,t)\,([\![t]\!]\,\mathrm{freshEnv}_{\Delta})$ for any term t, if we pick s as the identity substitution $\mathsf{id}_{\mathrm{S}} : \Delta \vdash_{\mathrm{S}} \Delta$, and δ as $\mathrm{freshEnv}_{\Delta} : [\![\Delta]\!]_{\Delta}$, since they can be shown to be related as $\mathrm{L}_{\Delta,\Delta}\,\mathsf{id}_{\mathrm{S}}\,\mathrm{freshEnv}_{\Delta}$. From this it follows that $\mathrm{subst}\,\mathsf{id}_{\mathrm{S}}\,t \sim \mathrm{reify}_A\,([\![t]\!]\,\mathrm{freshEnv}_{\Delta})$, and further that $t \sim \mathrm{quote}\,[\![t]\!]$ from

$$
\begin{aligned}
&\mathrm{L}_{A,\Gamma} : \Gamma \vdash A \to [\![A]\!]_\Gamma \to \mathrm{Type} \\
&\mathrm{L}_{\iota,\Gamma} \quad t\, n = t \sim \mathrm{quote}\, n \\
&\mathrm{L}_{A \Rightarrow B,\Gamma}\, t\, f = \forall \Gamma', o : \Gamma \leqslant \Gamma', u, a.\, \mathrm{L}_{A,\Gamma'}\, u\, a \to \mathrm{L}_{B,\Gamma'}\, (\mathsf{app}\, (\mathrm{wk}\, o\, t)\, u)\, (f\, o\, a) \\
&\mathrm{L}_{\Box A,\Gamma} \quad t\, b = \forall \Gamma', o : \Gamma \leqslant \Gamma', e : \Gamma' \lhd_{\lambda_{\mathrm{IK}}} \Delta.\, \mathrm{L}_{A,\Delta}\, (\mathsf{unbox}_{\lambda_{\mathrm{IK}}}\, (\mathrm{wk}\, o\, t)\, e)\, (b\, o\, e)
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{L}_{\Delta,\Gamma} : \Gamma \vdash_{\mathrm{S}} \Delta \to [\![\Delta]\!]_\Gamma \to \mathrm{Type} \\
&\mathrm{L}_{\cdot,\Gamma} \quad \mathsf{empty} \qquad\qquad\qquad\qquad () \quad = \top \\
&\mathrm{L}_{(\Delta,A),\Gamma}\, (\mathsf{ext}\, s\, t) \qquad\qquad\qquad (\delta, a) = \mathrm{L}_{\Delta,\Gamma}\, s\, \delta \times \mathrm{L}_{A,\Gamma}\, t\, a \\
&\mathrm{L}_{(\Delta,\blacksquare),\Gamma}\, (\mathsf{ext}_{\blacksquare}\, s\, (e : \Theta \lhd_{\lambda_{\mathrm{IK}}} \Gamma))\, (\delta, e) = \mathrm{L}_{\Delta,\Theta}\, s\, \delta
\end{aligned}
$$

Figure E.9.: Logical relations for λ_{IK}

the definition of quote and the fact that $\mathrm{subst}\, \mathsf{id}_{\mathrm{S}}\, t = t$. As a result, the composite norm $= \mathrm{quote} \circ [\![-]\!]$ is *adequate*, i.e. $\mathrm{norm}\, t = \mathrm{norm}\, t'$ implies $t \sim t'$.

The soundness of λ_{IK} with respect to possible-world models (see Theorem E.3.1) directly entails $\mathrm{quote}\, [\![t]\!] = \mathrm{quote}\, [\![u]\!] : \Gamma \vdash_{\mathrm{NF}} A$ for all terms t, $u : \Gamma \vdash A$ such that $\Gamma \vdash t \sim u : A$, which means that $\mathrm{norm} = \mathrm{quote} \circ [\![-]\!]$ is *complete*. Note that this terminology might be slightly confusing because it is the *soundness* of $[\![-]\!]$ that implies the *completeness* of norm.

Theorem E.3.2. *Let $\mathcal{M}$ denote the possible-world model over the frame given by the relations $\Gamma \leqslant \Gamma'$ and $\Delta \lhd_{\lambda_{IK}} \Gamma$ and the valuation $V_{\iota,\Gamma} = \Gamma \vdash_{NE} \iota$.*

There is a function $\mathrm{quote} : \mathcal{M}([\![\Gamma]\!], [\![A]\!]) \to \Gamma \vdash_{NF} A$ *such that the composite* $\mathrm{norm} = \mathrm{quote} \circ [\![-]\!] : \Gamma \vdash A \to \Gamma \vdash_{NF} A$ *from terms to normal forms of λ_{IK} is complete and adequate.*

E.3.2. Extending to the Calculus λ_{IS4}

Terms, Substitutions and Equational Theory

To define the intrinsically-typed syntax of λ_{IS4}, we first define the modal accessibility relation on contexts in Figure E.10.

$$
\mathsf{nil} : \Gamma \lhd_{\lambda_{\mathrm{IS4}}} \Gamma \qquad \frac{e : \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma}{\mathsf{var}\, e : \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma, A} \qquad \frac{e : \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma}{\mathsf{lock}\, e : \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma, \blacksquare}
$$

Figure E.10.: Modal accessibility relation on contexts (λ_{IS4})

If $\Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma$ then Γ is an extension of Δ with as many locks as needed. Note that, in contrast to λ_{IK}, the modal accessibility relation is both reflexive and transitive. This corresponds to the conditions on the accessibility relation for the logic IS4.

Figure E.11 presents the changes of λ_{IK} that yield λ_{IS4}. The terms are the same as λ_{IK} with the exception of Rule $\lambda_{\mathrm{IK}}/\Box$-Elim which now includes the modal accessibility relation for λ_{IS4}. Similarly, the substitution rule for contexts with locks now refers to $\lhd_{\lambda_{\mathrm{IS4}}}$.

$$\frac{\Delta \vdash t : \Box A \qquad e : \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma}{\Gamma \vdash \mathsf{unbox}_{\lambda_{\mathrm{IS4}}}\, t\, e : A}\ \lambda_{\mathrm{IS4}}/\Box\text{-Elim} \qquad \frac{\Theta \vdash s : \Delta \qquad e : \Theta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma}{\Gamma \vdash_{\mathrm{S}} \mathsf{ext}_{🔒}\, s\, e : \Delta, 🔒}$$

Figure E.11.: Intrinsically-typed terms and substitutions of λ_{IS4} (omitting the unchanged rules of Figure E.5)

Figure E.12 presents the equational theory of the modal fragment of λ_{IS4}. This is a slightly modified version of λ_{IK} (cf. Figure E.7) that accommodates the changes to the rule $\lambda_{\mathrm{IS4}}/\Box$-Elim. Unlike before, Rule $\Box$-β now performs a substitution to modify the term $\Delta, 🔒 \vdash t : A$ to a term of type $\Gamma \vdash A$. Note that the result of such a substitution need not yield the same term since substitution may change the context extension of some subterm.

$$\frac{\Delta, 🔒 \vdash t : A \qquad e : \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma}{\Gamma \vdash \mathsf{unbox}_{\lambda_{\mathrm{IS4}}}\, (\mathsf{box}\, t)\, e \sim \mathrm{subst}\, (\mathsf{ext}_{🔒}\, \mathsf{id}_{\mathrm{S}}\, e)\, t}\ \Box\text{-}\beta$$

$$\frac{\Gamma \vdash t : \Box A}{\Gamma \vdash t \sim \mathsf{box}\, (\mathsf{unbox}_{\lambda_{\mathrm{IS4}}}\, t\, (\mathsf{lock}\, \mathsf{nil}))}\ \Box\text{-}\eta$$

Figure E.12.: Equational theory for λ_{IS4} (omitting the unchanged rules of Figure E.7)

Possible-World Semantics

Giving possible-world semantics for λ_{IS4} requires an additional frame condition on the relation R_m: it must be reflexive and transitive. Evaluation proceeds as before, where we use a function $\text{trim}_{\lambda_{\text{IS4}}} : \forall w.\, [\![\Gamma]\!]_w \to \Delta \lhd_{\lambda_{\text{IS4}}} \Gamma \to [\![\Delta, \blacksquare]\!]_w$ to manipulate the environment for evaluating $\mathsf{unbox}_{\lambda_{\text{IS4}}}\, t\, e$, as seen below.

$$\begin{aligned}[\![\mathsf{unbox}_{\lambda_{\text{IS4}}}\, t\, e]\!]\, \gamma &= [\![t]\!]\, \delta\, \mathsf{id}_{\leqslant}\, m \\ &\text{where } (\delta, m) = \text{trim}_{\lambda_{\text{IS4}}}\, \gamma\, e\end{aligned}$$

The additional frame requirements ensures that the function $\text{trim}_{\lambda_{\text{IS4}}}$ can be implemented. For example, consider implementing the case of $\text{trim}_{\lambda_{\text{IS4}}}$ for some argument of type $[\![\Gamma]\!]_w$ and the extension $\mathsf{nil} : \Gamma \lhd_{\lambda_{\text{IS4}}} \Gamma$ that adds zero locks. The desired result is of type $[\![\Gamma, \blacksquare]\!]_w$, which is defined as $\sum_v [\![\Gamma]\!]_v \times v\ R_m\ w$. We construct such a result using the argument of $[\![\Gamma]\!]_w$ by picking v as w itself, and using the reflexivity of R_m to construct a value of type $w\ R_m\ w$. Similarly, the transitivity of R_m is required when the context extension adds more than one lock.

Analogously to Theorem E.3.1, we state the soundness of λ_{IS4} with respect to *reflexive and transitive* possible-world models before we instantiate it with the NbE model that we will construct in the next section. In addition to the coherence conditions stated before Theorem E.3.1 the soundness proof for λ_{IS4} relies on coherence conditions involving the additional proofs refl_{m} and trans_{m} that a reflexive and transitive modal accessibility relation R_m must come equipped with. Specifically, trans_{m} also needs to be associative, refl_{m} also needs to be a unit for trans_{m}, and the proofs of the factorization condition also need to satisfy the functoriality laws in the modal accessibility argument, i.e. $\text{factor}_{\text{i}}\, (\text{refl}_{\text{m}}\, w)\, i = i$, $\text{factor}_{\text{m}}\, (\text{refl}_{\text{m}}\, w)\, i = \text{refl}_{\text{m}}\, w'$, $\text{factor}_{\text{i}}\, (\text{trans}_{\text{m}}\, n\, m)\, i = \text{factor}_{\text{i}}\, n\, i'$ and $\text{factor}_{\text{m}}\, (\text{trans}_{\text{m}}\, n\, m)\, i = \text{trans}_{\text{m}}\, (\text{factor}_{\text{m}}\, n\, i')\, (\text{factor}_{\text{m}}\, m\, i)$ where $i' \coloneqq \text{factor}_{\text{i}}\, m\, i : w\ R_i\ w'$.

Proposition E.3.2. *Let $\mathcal{C}$ be a Cartesian closed category equipped with a comonad $\Box$ that has a left adjoint $\blacksquare \dashv \Box$, then equivalent terms t and $u : \Gamma \vdash A$ denote equal morphisms in $\mathcal{C}$.*

Proof. This is a version of Clouston (2018, Theorem 4.8) for λ_{IS4} where the side condition of Rule $\lambda_{\text{IS4}}/\Box$-Elim appears as an argument to the term former unbox and hence idempotency is not imposed on the comonad $\Box$. □

Theorem E.3.3. *Let $\mathcal{M}$ be a possible-world model (see Definition E.3.2) such that the modal accessibility relation R_m is reflexive and transitive. If two terms t and $u : \Gamma \vdash A$ of λ_{IS4} are equivalent (see Figure E.12) then the functions $[\![t]\!]$ and $[\![u]\!] : \forall w.\, [\![\Gamma]\!]_w \to [\![A]\!]_w$ as determined by $\mathcal{M}$ are equal.*

Proof. The right adjoint determined by a reflexive and transitive frame has a comonad structure so that we can conclude by applying Proposition E.3.2. □

NbE Model

The normal forms of λ_{IS4} are defined as before, except for the following rule replacing the neutral rule $\lambda_{\mathrm{IK}}/\textsc{Ne}/\Box\text{-}\textsc{Elim}$.

$$\frac{\Delta \vdash_{\mathrm{NE}} n : \Box A \qquad e : \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma}{\Gamma \vdash_{\mathrm{NE}} \mathsf{unbox}_{\lambda_{\mathrm{IS4}}}\, n\, e : A}\ \lambda_{\mathrm{IS4}}/\textsc{Ne}/\Box\text{-}\textsc{Elim}$$

The NbE model construction also proceeds in the same way, where we now pick the relation R_m as arbitrary extensions of a context: $\Delta\ R_m\ \Gamma = \Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma$. The modal fragment for reify and reflect are now implemented as follows:

$$\begin{aligned}
&\mathrm{reify}_{\Box A,\Gamma} \quad b = \mathsf{box}\,(\mathrm{reify}_{A,(\Gamma,🔒)}\,(b\,\mathsf{id}_{\leqslant}\,(\mathsf{lock}\,\mathsf{nil})))\\
&\mathrm{reflect}_{\Box A,\Gamma}\, n = \lambda(o : \Gamma \leqslant \Gamma').\, \lambda(e : \Gamma' \lhd_{\lambda_{\mathrm{IS4}}} \Delta).\, \mathrm{reflect}_{A,\Delta}\,(\mathsf{unbox}\,(\mathrm{wk}\, o\, n)\, e)
\end{aligned}$$

Theorem E.3.4. *Let $\mathcal{M}$ denote the possible-world model over the reflexive and transitive frame given by the relations $\Gamma \leqslant \Gamma'$ and $\Delta \lhd_{\lambda_{IS4}} \Gamma$ and the valuation $V_{\iota,\Gamma} = \Gamma \vdash_{NE} \iota$.*

There is a function quote : $\mathcal{M}([\![\Gamma]\!], [\![A]\!]) \to \Gamma \vdash_{NF} A$ *such that the composite* norm = quote $\circ\ [\![-]\!] : \Gamma \vdash A \to \Gamma \vdash_{NF} A$ *from terms to normal forms of λ_{IS4} is complete and adequate.*

The proof of this Theorem requires us to identify terms by extending the equational theory of λ_{IS4} with an additional rule. To understand the need for it, consider unboxing a term $\Gamma \vdash t : \Box A$ into an extended context Γ, B in λ_{IS4}. We may first weaken t as $\Gamma, B \vdash \mathrm{wk}\,(\mathsf{drop}\,\mathsf{id}_{\leqslant})\, t : \Box A$ and then apply unbox as $\Gamma, B \vdash \mathsf{unbox}\,(\mathrm{wk}\,(\mathsf{drop}\,\mathsf{id}_{\leqslant})\, t)\,\mathsf{nil} : A$. However, we may also apply unbox on t as $\Gamma, B \vdash \mathsf{unbox}\, t\,(\mathsf{var}\,\mathsf{nil}) : A$. This weakens the term "explicitly" in the sense that the weakening with B is recorded in the term by the proof $\mathsf{var}\,\mathsf{nil}$ of the modal accessibility relation $\Gamma \lhd_{\lambda_{\mathrm{IS4}}} \Gamma, B$. The two ways of unboxing $\Gamma \vdash t : \Box A$ into the extended context Γ, B result in two terms with the same denotation in the possible-world semantics but *distinct* typing derivations. We wish the two typing derivations $\mathsf{unbox}\, t\,(\mathsf{var}\,\mathsf{nil})$ and $\mathsf{unbox}\,(\mathrm{wk}\,(\mathsf{drop}\,\mathsf{id}_{\leqslant})\, t)\,\mathsf{nil}$ to be identified. For this reason, we extend the equational theory of λ_{IS4} with the rule $\mathsf{unbox}\, t\,(\mathrm{trans_m}\, e\, e') \sim \mathsf{unbox}\,(\mathrm{wk}\,(\mathrm{toOPE}e)\, t)\, e'$ for any *lock-free* extension e, which can be converted to a sequence of drops using the function toOPE. Explicit weakening can also be avoided by, instead of extending the equational theory, changing the definition of the modal accessibility relation such that

$\Delta \lhd_{\lambda_{\mathrm{IS4}}} \Gamma$ holds only if $\Gamma = \Delta$ or $\Gamma = \Delta, 🔒, \Gamma'$ for some Γ'. Note that the modal accessibility relation for λ_{IK}, where the issue of explicit weakening does not occur, satisfies this property.

E.3.3. Extending to the Calculi $\lambda_{\mathbf{IT}}$ and $\lambda_{\mathbf{IK4}}$

The NbE model construction for λ_{IT} and λ_{IK4} follows a similar pursuit as λ_{IS4}. We define suitable modal accessibility relations $\lhd_{\lambda_{\mathrm{IT}}}$ and $\lhd_{\lambda_{\mathrm{IK4}}}$ as extensions that allow the addition of at most one 🔒, and at least one lock 🔒, respectively. To give possible-world semantics, we require an additional frame condition that the relation R_m be reflexive for λ_{IT} and transitive for λ_{IK4}. For evaluation, we use a function $\mathrm{trim}_{\lambda_{\mathrm{IT}}} : [\![\Gamma]\!]_w \to \Delta \lhd_{\lambda_{\mathrm{IT}}} \Gamma \to [\![\Delta, 🔒]\!]_w$ for λ_{IT}, and similarly $\mathrm{trim}_{\lambda_{\mathrm{IK4}}}$ for λ_{IK4}. The modification to the neutral rule $\lambda_{\mathrm{IK}}/\textsc{Ne}/\Box\text{-}\textsc{Elim}$ is achieved as before in λ_{IS4} using the corresponding modal accessibility relations. Unsurprisingly, reification and reflection can also be implemented, thus yielding normalization functions for both λ_{IT} and λ_{IK4}.

E.4. Completeness, Decidability and Logical Applications

In this section we record some immediate consequences of the model constructions we presented in the previous section.

Completeness of the Equational Theory As a corollary of the adequacy of an NbE model $\mathcal{N}$, i.e. $\Gamma \vdash t \sim u : A$ whenever $[\![t]\!] = [\![u]\!] : \mathcal{N}([\![\Gamma]\!], [\![A]\!])$, we obtain completeness of the equational theory with respect to the class of models that the respective NbE model belongs to. Given the NbE models constructed in Sections E.3.1 and E.3.2 this means that the equational theories of λ_{IK} and λ_{IS4} (cf. Figure E.7) are (sound and) complete with respect to the class of Cartesian closed categories equipped with an adjunction and a right-adjoint comonad, respectively.

Theorem E.4.1. *Let $t, u : \Gamma \vdash A$ be two terms of λ_{IK}. If for all Cartesian closed categories $\mathcal{M}$ equipped with an adjunction it is the case that $[\![t]\!] = [\![u]\!] : \mathcal{M}([\![\Gamma]\!], [\![A]\!])$ then $\Gamma \vdash t \sim u : A$.*

Proof. Let $\mathcal{M}_0$ be the model we constructed in Section E.3.1. Since $\mathcal{M}_0$ is a Cartesian closed category equipped with an adjunction, by assumption we have $[\![t]\!]_{\mathcal{M}_0} = [\![u]\!]_{\mathcal{M}_0}$. And lastly, since $\mathcal{M}_0$ is an NbE model, we have $\Gamma \vdash t \sim \mathrm{quote}\, [\![t]\!]_{\mathcal{M}_0} = \mathrm{quote}\, [\![u]\!]_{\mathcal{M}_0} \sim u : A$. □

Note that this statement corresponds to Clouston (2018, Theorem 3.2) but there it is obtained via a term model construction and for the term model to be equipped with an adjunction the calculus needs to be first extended with an internalization of the operation 🔒 on contexts as an operation ♦ on types.

Theorem E.4.2. *Let $t, u : \Gamma \vdash A$ be two terms of λ_{IS4}. If for all Cartesian closed categories $\mathcal{M}$ equipped with a right-adjoint comonad it is the case that $[\![t]\!] = [\![u]\!] : \mathcal{M}([\![\Gamma]\!], [\![A]\!])$ then $\Gamma \vdash t \sim u : A$.*

Proof. As for Theorem E.4.1. □

This statement corresponds to Clouston (2018, Section 4.4) but there it is proved for an equational theory that identifies terms up to differences in the accessibility proofs and with respect to the class of models where the comonad is *idempotent*, to which the model of Section E.3.2 does not belong.

Completeness of the Deductive Theory Using the quotation function of an NbE model $\mathcal{N}$, i.e. quote : $\mathcal{N}([\![\Gamma]\!], [\![A]\!]) \to \Gamma \vdash A$, we obtain completeness of the deductive theory with respect to the class of models that the respective NbE model belongs to. Given the NbE models constructed in Sections E.3.1 and E.3.2 this means that the deductive theories of λ_{IK} and λ_{IS4} (cf. Figures E.5 and E.2) are (sound and) complete with respect to the class of possible-world models with an arbitrary frame and a reflexive–transitive frame, respectively.

Theorem E.4.3. *Let Γ : Ctx be a context and A : Ty a type. If for all possible-world models $\mathcal{M}$ it is the case that $\mathcal{M}([\![\Gamma]\!], [\![A]\!])$ is inhabited then there is a term $t : \Gamma \vdash A$ of λ_{IK}.*

Proof. Let $\mathcal{M}_0$ be the model we constructed in Section E.3.1. Since $\mathcal{M}_0$ is a possible-world model, by assumption we have a morphism $p : \mathcal{M}_0([\![\Gamma]\!], [\![A]\!])$. And lastly, since $\mathcal{M}_0$ is an NbE model, we have the term quote $p : \Gamma \vdash A$. □

Theorem E.4.4. *Let Γ : Ctx be a context and A : Ty a type. If for all possible-world models $\mathcal{M}$ with a reflexive–transitive frame it is the case that $\mathcal{M}([\![\Gamma]\!], [\![A]\!])$ is inhabited then there is a term $t : \Gamma \vdash A$ of λ_{IS4}.*

Proof. As for Theorem E.4.3. □

Note that the proofs of Theorems E.4.3 and E.4.4 are constructive.

Decidability of the Equational Theory As a corollary of the completeness and adequacy of an NbE model $\mathcal{N}$, i.e. $\Gamma \vdash t \sim u : A$ if and only if $[\![t]\!] = [\![u]\!] : \mathcal{N}([\![\Gamma]\!], [\![A]\!])$, we obtain decidability of the equational theory from decidability of the equality of normal forms $n,\ m : \Gamma \vdash_{\text{NF}} A$. Given the NbE models constructed in Sections E.3.1 and E.3.2 this means that the equational theories of λ_{IK} and λ_{IS4} (cf. Figure E.7) are decidable.

To show that any of the following decision problems $P(x)$ is decidable we give a *constructive* proof of the proposition $\forall x.\, P(x) \vee \neg P(x)$. Such a proof can be understood as the construction of an algorithm d that takes as input an x and produces as output a Boolean $d(x)$, alongside a correctness proof that $d(x)$ is true if and only if $P(x)$ holds.

Theorem E.4.5. *For any two terms $t,\ u : \Gamma \vdash A$ of λ_{IK} the problem whether $t \sim u$ is decidable.*

Proof. We first observe that for any two normal forms $n,\ m : \Gamma \vdash_{\text{NF}} A$ of λ_{IK} the problem whether $n = m$ is decidable by proving $\forall n, m.\, n = m \vee n \neq m$ constructively. All the cases of an simultaneous induction on $n,\ m : \Gamma \vdash_{\text{NF}} A$ are immediate.

Let $\mathcal{N}$ be the NbE model we constructed in Section E.3.1. Completeness and adequacy of $\mathcal{N}$ imply that we have $t \sim u$ if and only if $\operatorname{norm} t = \operatorname{norm} u$ for the function $\operatorname{norm} : \Gamma \vdash A \to \Gamma \vdash_{\text{NF}} A, t \mapsto \operatorname{quote} [\![t]\!]$. Now, $t \sim u$ is decidable because $\operatorname{norm} t = \operatorname{norm} u$ is decidable by the observation we started with. □

Theorem E.4.6. *For any two terms $t,\ u : \Gamma \vdash A$ of λ_{IS4} the problem whether $t \sim u$ is decidable.*

Proof. As for Theorem E.4.5. □

Denecessitation The last of the consequences of the NbE model constructions we record is of a less generic flavour than the other three, namely it is an application of normal forms to a basic proof-theoretic result in modal logic.

Using invariance of truth in possible-world models under bisimulation[2] it can be shown that $\Box A$ is a valid formula of IK (or IS4) if and only if A is. A completeness theorem then implies the same for provability of $\Box A$ and A. The statement for proofs in λ_{IK} (and λ_{IS4}) can also be shown by inspection of normal forms as follows.

[2] Invariance of truth under bisimulation says that if w and v are two bisimilar worlds in two possible-world models $\mathcal{M}_0$ and $\mathcal{M}_1$, respectively, then for all formulas A it is the case that $[\![A]\!]_w$ holds in $\mathcal{M}_0$ if and only if $[\![A]\!]_v$ does in $\mathcal{M}_1$.

Firstly, we note that while deduction is not closed under arbitrary context extensions (including locks) it is closed under extensions (including locks) *on the left*:

Lemma E.4.1 (cf. Clouston (2018, Lemma A.1))**.** *Let* Δ, $\Gamma : \mathrm{Ctx}$ *be arbitrary contexts, both possibly containing locks, and* $A : \mathrm{Ty}$ *an arbitrary type. There is an operation* $\Gamma \vdash A \to \Delta, \Gamma \vdash A$ *on terms of* λ_{IK} *(and* λ_{IS4}*), where* Δ, Γ *denotes context concatenation.*

Proof. By recursion on terms. □

And, secondly, we note that also a converse of this Lemma holds by inspection of normal forms:

Lemma E.4.2. *Let* Δ, $\Gamma : \mathrm{Ctx}$ *be arbitrary contexts, both possibly containing locks,* $A : \mathrm{Ty}$ *an arbitrary type and* $t : \Delta, \Gamma \vdash A$ *a term of* λ_{IK} *(or* λ_{IS4}*) in the concatenated context* Δ, Γ *that does* not *mention any variables from* Δ*, then there is a term* $t' : \Gamma \vdash A$ *of* λ_{IK} *(or* λ_{IS4}*, respectively).*

Proof. Since normalization (see Theorems E.3.2 and E.3.4) does not introduce new free variables it suffices to prove the statement for terms in normal form. We do so by induction on normal forms $n : \Delta, \Gamma \vdash_{\mathrm{NF}} A$ (see Figure E.8). The only nonimmediate step is for n of the form $\mathsf{unbox}\, n'\, e$ for some neutral element $n' : \Delta' \vdash_{\mathrm{NE}} \Box A$ and $\Delta' \lhd \Delta \leqslant \Delta, \Gamma$. But in that case the induction hypothesis says that we have a neutral element $n'' : \cdot \vdash_{\mathrm{NE}} \Box A$, which is impossible. □

Note that some form of normalization seems to be needed in the proof of Lemma E.4.2. More specifically, the "strengthening" of a term of the form $\mathsf{unbox}\, t\, e$ from the context $\cdot, 🔒, \cdot$ to the empty context $\cdot$ cannot possibly result in a term of the form $\mathsf{unbox}\, t'\, e'$ because there is *no* context Γ such that $\Gamma \lhd \cdot$ in λ_{IK}. As an example, consider the term $\mathsf{unbox}\, (\mathsf{box}\, (\lambda\, x.\, x))\, \mathsf{nil}$, which needs to be strengthened to $\lambda\, x.\, x$.

With these two Lemmas at hand we are ready to prove denecessitation through normalization:

Theorem E.4.7. *Let* $A : \mathrm{Ty}$ *be an arbitrary type. There is a term* $t : \cdot \vdash A$ *of* λ_{IK} *(or* λ_{IS4}*) if and only if there is a term* $u : \cdot \vdash \Box A$ *of* λ_{IK} *(or* λ_{IS4}*, respectively), where* $\cdot : \mathrm{Ctx}$ *denotes the* empty *context.*

Proof. From a term $t : \cdot \vdash A$ we can construct a term $t' : \cdot, 🔒 \vdash A$ using Lemma E.4.1 and thus the term $u = \mathsf{box}\, t' : \cdot \vdash \Box A$.

In the other direction, from a term $u : \cdot \vdash \Box A$ we obtain a normal form $u' = \operatorname{norm} u : \cdot \vdash_{\mathrm{NF}} \Box A$ using Theorems E.3.2 and E.3.4. By inspection of normal forms (see Figure E.8) we know that u' must be of the form $\mathsf{box}\, v$ for some normal form $v : \cdot, \blacksquare \vdash_{\mathrm{NF}} A$, from which we obtain a term $t : \cdot \vdash A$ using Lemma E.4.2 since the context $\cdot, \blacksquare$ does not declare any variables that could have been mentioned in v. □

This concludes this section on some consequences of the model constructions presented in this paper. Note that the consequences we recorded are completely independent of the concrete model construction. To wit, the two completeness theorems follow from the mere existence of an NbE model, and the decidability and denecessitation theorems follow from the mere existence of a normalization function.

E.5. Programming-Language Applications

In this section, we discuss some implications of normalization for Fitch-style calculi for specific interpretations of the necessity modality in the context of programming languages. In particular, we show how normalization can be used to prove properties about program calculi by leveraging the shape of normal forms of terms. We extend the term calculi presented earlier with application-specific primitives, ensure that the extended calculi are in fact normalizing, and then use this result to prove properties such as capability safety, noninterference, and binding-time correctness. Note that we do not mechanize these results in Agda and do not prove these properties in their full generality, but only illustrate special cases. Although possible, proving the general properties requires further technical development that obscures the main idea underlying the use of normal forms for simplifying these proofs.

E.5.1. Capability Safety

Choudhury and Krishnaswami (2020) present a modal type system based on IS4 for a programming language with implicit effects in the style of ML (Milner et al. 1990) and the computational lambda calculus (Moggi 1989). In this language, programs need access to capabilities to perform effects. For instance, a primitive for printing a string requires a capability as an argument in addition to the string to be printed. Crucially, capabilities cannot be introduced within the language, and must be obtained either from the global context (called *ambient* capabilities) or as a function argument.

Let us denote the type of capabilities by Cap. Passing a printing capability c to a function of type $\mathsf{Cap} \Rightarrow \mathsf{Unit}$ in a language that uses capabilities to print yields a program that either 1. does not print, 2. prints using only the capability c, or 3. prints using ambient capabilities (and possibly c). A program that at most uses the capabilities that it is passed explicitly, as in the cases 1 and 2, is said to be *capability safe*. To identify such programs, Choudhury and Krishnaswami (2020) introduce a comonadic modality $\Box$ to capture capability safety. Their type system is loosely based on the dual-context calculus for IS4 (Pfenning and Davies 2001; Kavvos 2020). A term of type $\Box A$ is enforced to be capability safe by making the introduction rule for $\Box$ "brutally" remove all capabilities from the typing context. As a result, programs with the type $\Box(\mathsf{Cap} \Rightarrow \mathsf{Unit})$ are denied ambient capabilities and thus guaranteed to behave like the cases 1 and 2.

Choudhury and Krishnaswami (2020) characterize capability safety precisely using their *capability space* model. A capability space (X, w_X) is a set X and a weight relation w_X that assigns sets of capabilities to every member in X. In this model, they define a comonad that restricts the underlying set of a capability space to those elements that are only related to the empty set of capabilities. This comonad has a left adjoint that replaces the weight relation of a capability space by the relation that relates every element to the empty set of capabilities. This adjunction suggests that capability spaces are a model of λ_{IS4} and we may thus use λ_{IS4} to write programs that support reasoning about capability safety.

In this section, we present a calculus λ_{IS4}+Moggi$^{\mathrm{Cap}}$ that extends λ_{IS4} with a capability type and a monad for printing effects. We extend the normalization algorithm for λ_{IS4} to λ_{IS4}+Moggi$^{\mathrm{Cap}}$ and show that the resulting normal forms can be used to prove a kind of capability safety. In contrast to the language presented by Choudhury and Krishnaswami (2020), λ_{IS4}+Moggi$^{\mathrm{Cap}}$ models a language where effects are explicit in the type of a term. Languages with explicit effects, such as Haskell (Augustsson et al. 1990) (with the IO monad) or PureScript (Freeman 2013) (with the `Effect` monad), can also benefit from a mechanism for capability safety, and we begin with an example in a hypothetical extension of PureScript to illustrate this.

Example in PureScript Let us consider web development in PureScript. A web application may consist of a mashup of several components, e.g. social media, news feed, or chat, provided by untrusted sources. A component is a function of type

```
type Component = Element -> Effect Unit
```

that takes as a parameter the DOM element where the component will be rendered. For the correct functioning of the web application, it is important that components do not interfere with each other in malicious ways. For example, a malicious component (of Bob) could illegitimately overwrite a DOM element (of Alice):

```
evilBob :: Component
evilBob e = do w      <- window
               doc    <- document w
               aliceE <- getElementById "alice.app" doc
               setTextContent "Alice has been hacked!" aliceE
```

The issue here is that Bob has unrestricted access to the function `window :: Effect Window`, and is able to obtain the DOM using `document :: Window -> Effect DOM` and overwrite an element that belongs to Alice. Capabilities can be leveraged to restrict the access to `window`. We can achieve this by extending PureScript with a type `WindowCap`, a type constructor `Box` that works similarly to Choudhury and Krishnaswami's $\Box$, and replacing the function `window` with a function `window' :: WindowCap -> Effect Window` that requires an additional capability argument. By making `WindowCap` an ambient capability that is available globally, all existing programs retain their unrestricted access to retrieve a window as before. The difference now, however, is that we can selectively restrict some programs and limit their access to `WindowCap` using `Box`. We can define a variant of the type `Component` as:

```
type Component' = Box (Element -> Effect Unit)
```

By requiring Bob to write a component of the type `Component'`, we are ensured that Bob cannot overwrite an element that belongs to Alice. This is because the `Box` type constructor used to define `Component'` disallows access to all ambient capabilities (including `WindowCap`), and thus restricts Bob to only using the given `Element` argument. In particular, the program `evilBob` cannot be reproduced with the type `Component'` since the substitute function `window'` requires a capability that is neither available as an argument nor as an ambient capability.

Extension with a Capability and a Monad We extend λ_{IS4} with a monad for printing based on Moggi's monadic metalanguage (Moggi 1991). We introduce a type $\mathsf{T}\,A$ that denotes a monadic computation that can print before returning a value of type A, a type Cap for capabilities, and a type String for strings. Figure E.13 summarizes the terms that correspond to this extension. The term

$$\text{Ty} \quad A, B ::= \ldots \mid \mathsf{T}\, A \mid \mathsf{Cap} \mid \mathsf{String} \mid \mathsf{Unit} \qquad \text{Ctx} \quad \Gamma ::= \ldots$$

$$\textsf{T-Intro}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{return}\, t : \mathsf{T}\, A} \qquad \textsf{T-Elim}\ \frac{\Gamma \vdash t : \mathsf{T}\, A \qquad \Gamma, A \vdash u : \mathsf{T}\, B}{\Gamma \vdash \mathsf{let}\, t\, u : \mathsf{T}\, B}$$

$$\textsf{Unit-Intro}\ \Gamma \vdash \mathsf{unit} : \mathsf{Unit} \qquad \textsf{String-Lit}\ \frac{}{\Gamma \vdash \mathsf{str}_s : \mathsf{String}}\ s \in \mathrm{String}$$

$$\textsf{T-Print}\ \frac{\Gamma \vdash c : \mathsf{Cap} \qquad \Gamma \vdash s : \mathsf{String}}{\Gamma \vdash \mathsf{print}\, c\, s : \mathsf{T}\, \mathsf{Unit}}$$

Figure E.13.: Types, contexts and terms of λ_{IS4}+Moggi$^{\mathrm{Cap}}$ (omitting the unchanged rules of Figures E.5 and E.11)

construct print is used for printing. The equational theory of λ_{IS4}+Moggi$^{\mathrm{Cap}}$ and the corresponding normal forms are summarized in Figure E.14 and Figure E.15, respectively.

To extend the NbE model of λ_{IS4} with an interpretation for the monad, we use the standard techniques used for normalizing computational effects (Ahman and Staton 2013; Filinski 2001). The interpretation of the other primitive types also follows a standard pursuit (Valliappan, Russo, et al. 2021): we interpret Cap by neutrals of type Cap and String by the disjoint union of String and neutrals of type String. The difference in their interpretation is caused by the fact that there is no introduction form for the type Cap.

Proving Capability Safety Programs that lack access to capabilities are necessarily capability safe. We say that a program $\Gamma \vdash p : A$ is *trivially capability safe* if there is a program $\cdot \vdash p' : A$ such that $\Gamma \vdash p \sim \mathrm{leftConcat}_\Gamma\, p' : A$, where $\mathrm{leftConcat}_\Gamma : \forall \Delta, A.\, \Delta \vdash A \to \Gamma, \Delta \vdash A$ can be defined similarly to the operation given by Lemma E.4.1 for λ_{IS4}.

First, we prove an auxiliary Lemma about normal forms with a capability in context.

Lemma E.5.1. *For any context Γ, type A and normal form $c : \mathsf{Cap}, \blacksquare, \Gamma \vdash_{NF} n : A$ there is a normal form $\cdot, \blacksquare, \Gamma \vdash_{NF} n' : A$ such that $n = \mathrm{leftConcat}_{c:\mathsf{Cap}}\, n'$.*

Proof. We prove the statement for both normal forms and neutral elements

$$\mathsf{T}\text{-}\beta\ \frac{\Gamma \vdash t : A \qquad \Gamma, A \vdash u : \mathsf{T}\, B}{\Gamma \vdash \mathsf{let}\,(\mathsf{return}\, t)\, u \sim \mathrm{subst}\,(\mathsf{ext}\,\mathsf{id}_{\mathrm{s}}\, t)\, u} \qquad \mathsf{T}\text{-}\eta\ \frac{\Gamma \vdash t : \mathsf{T}\, A}{\Gamma \vdash t \sim \mathsf{let}\, t\,(\mathsf{return}\,(\mathsf{var}\,\mathsf{zero}))}$$

$$\mathsf{T}\text{-}\gamma\ \frac{\Gamma \vdash t_1 : A \qquad \Gamma, A \vdash t_2 : \mathsf{T}\, B \qquad \Gamma, B \vdash t_3 : \mathsf{T}\, C}{\Gamma \vdash \mathsf{let}\,(\mathsf{let}\, t_1\, t_2)\, t_3 \sim \mathsf{let}\, t_1\,(\mathsf{let}\, t_2\,(\mathrm{wk}\,(\mathsf{keep}\,(\mathsf{drop}\,\mathsf{id}_{\leqslant}))\, t_3))}$$

Figure E.14.: Equational theory for λ_{IS4}+Moggi$^{\mathrm{Cap}}$ (omitting the unchanged rules of Figures E.7 and E.12)

by mutual induction. The only nonimmediate case is when the neutral is of the form $c : \mathsf{Cap}, \mathbb{\color{black}🔒}, \Gamma \vdash_{\mathrm{NE}} \mathsf{unbox}\, n\, e : A$ for some $n : \Delta \vdash_{\mathrm{NE}} \Box A$ and $e : \Delta \lhd_{\lambda_{\mathrm{IS4}}} c : \mathsf{Cap}, \mathbb{\color{black}🔒}, \Gamma$. We observe that there are no neutral elements of type $\Box A$ in context $c : \mathsf{Cap}$ and that hence Δ must contain the leftmost lock in $c : \mathsf{Cap}, \mathbb{\color{black}🔒}, \Gamma$. Thus, this case also holds by induction hypothesis. □

Now, we observe that all terms $c : \mathsf{Cap} \vdash t : \Box A$ are trivially capability safe. By normalization, we have that $c : \mathsf{Cap} \vdash t \sim \mathrm{norm}\, t : \Box A$. Given the definition of normal forms of λ_{IS4}+Moggi$^{\mathrm{Cap}}$, $\mathrm{norm}\, t$ must be $\mathsf{box}\, n$ for some normal form $c : \mathsf{Cap}, \mathbb{\color{black}🔒} \vdash_{\mathrm{NF}} n : A$. By Lemma E.5.1, there is a normal form $\mathbb{\color{black}🔒} \vdash_{\mathrm{NF}} n' : A$ such that $n = \mathrm{leftConcat}._{,\mathsf{Cap}}\, n'$. Since the operation leftConcat commutes with box, i.e. $\mathrm{leftConcat}._{,\mathsf{Cap}}\,(\mathsf{box}\, n') = \mathsf{box}\,(\mathrm{leftConcat}._{,\mathsf{Cap}}\, n')$, we also have that $t \sim \mathsf{box}\, n = \mathrm{leftConcat}._{,\mathsf{Cap}}\,(\mathsf{box}\, n')$. As a result, t must be trivially capability safe.

A consequence of this observation is that any term $c : \mathsf{Cap} \vdash t : \Box(\mathsf{T}\,\mathsf{Unit})$ is trivially capability safe. This means that t does not print since it could not possibly do so without a capability. Going further, we can also observe that $t \sim \mathsf{box}\,(\mathsf{return}\,\mathsf{unit}) : \Box(\mathsf{T}\,\mathsf{Unit})$, since the only normal form of type $\mathsf{T}\,\mathsf{Unit}$ in the empty context is $\cdot \vdash_{\mathrm{NF}} \mathsf{return}\,\mathsf{unit} : \mathsf{T}\,\mathsf{Unit}$. Note that this argument (and the one above) readily adapts to a vector of capabilities $\vec{c}$ in context as opposed to a single capability c.

E.5.2. Information-Flow Control

Information-flow control (IFC) (Sabelfeld and Myers 2003) is a technique used to protect the confidentiality of data in a program by tracking the flow of information within the program.

$$\textsc{Nf/T-Intro}\ \frac{\Gamma \vdash_{\text{NF}} m : A}{\Gamma \vdash_{\text{NF}} \mathsf{return}\, m : \mathsf{T}\, A} \qquad \textsc{Nf/T-Elim}\ \frac{\Gamma \vdash_{\text{NE}} n : \mathsf{T}\, A \qquad \Gamma, A \vdash_{\text{NF}} m : \mathsf{T}\, B}{\Gamma \vdash_{\text{NF}} \mathsf{let}\, n\, m : \mathsf{T}\, B}$$

$$\textsc{Nf/Unit-Intro}\ \Gamma \vdash_{\text{NF}} \mathsf{unit} : \mathsf{Unit} \qquad \textsc{Nf/Up-Cap}\ \frac{\Gamma \vdash_{\text{NE}} n : \mathsf{Cap}}{\Gamma \vdash_{\text{NF}} \mathsf{up}\, n : \mathsf{Cap}} \qquad \textsc{Nf/Up-String}\ \frac{\Gamma \vdash_{\text{NE}} n : \mathsf{String}}{\Gamma \vdash_{\text{NF}} \mathsf{up}\, n : \mathsf{String}}$$

$$\textsc{Nf/String-Lit}\ \frac{}{\Gamma \vdash_{\text{NF}} \mathsf{str}_s : \mathsf{String}}\ s \in \mathrm{String}$$

$$\textsc{Nf/T-Print}\ \frac{\Gamma \vdash_{\text{NF}} c : \mathsf{Cap} \quad \Gamma \vdash_{\text{NF}} s : \mathsf{String} \quad \Gamma, \mathsf{Unit} \vdash_{\text{NF}} m : \mathsf{T}\, A}{\Gamma \vdash_{\text{NF}} \mathsf{let}\, (\mathsf{print}\, c\, s)\, m : \mathsf{T}\, A}$$

Figure E.15.: Normal forms of λ_{IS4}+Moggi$^{\text{Cap}}$ (omitting the unchanged normal forms of λ_{IS4})

In type-based *static* IFC (e.g. Abadi et al. 1999; Shikuma and Igarashi 2008; Russo et al. 2008) types are used to associate values with confidentiality levels such as *secret* or *public*. The type system ensures that secret inputs do not interfere with public outputs, enforcing a security policy that is typically formalized as a kind of *noninterference* property (Goguen and Meseguer 1982).

Noninterference is proved by reasoning about the semantic behaviour of a program. Tomé Cortiñas and Valliappan (2019) present a proof technique that uses normalization for showing noninterference for a static IFC calculus based on Moggi's monadic metalanguage (Moggi 1991). This technique exploits the insight that normal forms represent equivalence classes of terms identified by their semantics, and thus reasoning about normal forms of terms (as opposed to terms themselves) vastly reduces the set of programs that we must take into consideration. Having developed normalization for Fitch-style calculi, we can leverage this technique to prove noninterference.

In this section, we extend λ_{IK} with Booleans (denoted λ_{IK}+Bool), extend the NbE model of λ_{IK} to λ_{IK}+Bool, and illustrate the technique of Tomé Cortiñas and Valliappan on λ_{IK}+Bool for proving noninterference. We interpret the type $\Box A$ as a secret of type A, and other types as public.

Extension with Booleans Noninterference can be better appreciated in the presence of a type whose values are distinguishable by an external observer. To this extent, we extend λ_{IK} with a type Bool and corresponding introduction and elimination forms—as described in Figure E.16.

$$\text{Ty} \quad A, B ::= \ldots \mid \mathsf{Bool} \qquad\qquad \text{Ctx} \quad \Gamma ::= \ldots$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ \textsf{Bool-Intro-true} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}\ \textsf{Bool-Intro-false}$$

$$\frac{\Gamma \vdash b : \mathsf{Bool} \qquad \Gamma, \Gamma' \vdash t_1 : A \qquad \Gamma, \Gamma' \vdash t_2 : A}{\Gamma, \Gamma' \vdash \mathsf{ifte}\, b\, t_1\, t_2 : A}\ \textsf{Bool-Elim}$$

Figure E.16.: Types, contexts and intrinsically-typed terms of λ_{IK}+Bool (omitting the unchanged rules of Figure E.5)

We modify the usual elimination rule for Bool by allowing the context of the conclusion $\mathsf{ifte}\, b\, t_1\, t_2$ and branches t_1 and t_2 in the rule Bool-Elim to extend the context of the scrutinee b. This modification (following Clouston (2018, Fig. 2)) enables the following *commuting conversion*, which is required to ensure that terms can be fully normalized and normal forms enjoy the subformula property:

$$\frac{\Delta \vdash b : \mathsf{Bool} \qquad \Delta, \Delta' \vdash t_1 : \Box A \qquad \Delta, \Delta' \vdash t_2 : \Box A \qquad e : \Delta, \Delta' \lhd \Gamma}{\Gamma \vdash \mathsf{unbox}\,(\mathsf{ifte}\, b\, t_1\, t_2)\, e \sim \mathsf{ifte}\, b\, (\mathsf{unbox}\, t_1\, e)\, (\mathsf{unbox}\, t_2\, e)}$$

A commuting conversion is required as usual for every other elimination rule, including the rule $\Rightarrow$-Elim. These are however standard and thus omitted here.

We extend the equational theory of λ_{IK} to λ_{IK}+Bool by adding the usual rules $\mathsf{ifte}\,\mathsf{true}\, t_1\, t_2 \sim t_1$, $\mathsf{ifte}\,\mathsf{false}\, t_1\, t_2 \sim t_2$, and $t \sim \mathsf{ifte}\, t\, \mathsf{true}\, \mathsf{false}$ for terms t of type Bool. The normal forms of λ_{IK}+Bool include those of λ_{IK} in addition to the following.

$$\frac{}{\Gamma \vdash_{\mathrm{NF}} \mathsf{true} : \mathsf{Bool}}\ \textsf{Nf/Bool-Intro-true} \qquad \frac{}{\Gamma \vdash_{\mathrm{NF}} \mathsf{false} : \mathsf{Bool}}\ \textsf{Nf/Bool-Intro-false}$$

$$\frac{\Gamma \vdash_{\mathrm{NE}} n : \mathsf{Bool} \quad \Gamma, \Gamma' \vdash_{\mathrm{NF}} m_1 : A \quad \Gamma, \Gamma' \vdash_{\mathrm{NF}} m_2 : A}{\Gamma, \Gamma' \vdash_{\mathrm{NF}} \mathsf{ifte}\, n\, m_1\, m_2 : A}\ \textsf{Nf/Bool-Elim}$$

Observe that a neutral of type Bool is not immediately in normal form, and must be expanded as $\mathsf{ifte}\, n\, \mathsf{true}\, \mathsf{false}$. This is unlike neutrals of the type ι, which are in normal form by Rule Nf/Up.

To extend the NbE model of λ_{IK} with Booleans, we leverage the interpretation of sum types used by Abel and Sattler (2019), who attribute their idea to Altenkirch and Uustalu (2004). This interpretation readily supports commuting conversions, and a minor refinement that reflects the change to the rule Bool-Elim yields a reifiable interpretation for Booleans in λ_{IK}+Bool.

Proving Noninterference A program $\cdot \vdash f : \Box A \Rightarrow \mathsf{Bool}$ is *noninterferent* if it is the case that $\cdot \vdash \mathsf{app}\, f\, s_1 \sim \mathsf{app}\, f\, s_2 : \mathsf{Bool}$ for any two secrets $\cdot \vdash s_1, s_2 : \Box A$. By instantiating A to Bool, we can show that any program $\cdot \vdash f : \Box\mathsf{Bool} \Rightarrow \mathsf{Bool}$ is noninterferent and thus cannot leak a secret Boolean argument. In λ_{IK}+Bool, the type system ensures that data of type $\Box A$ type can only influence (or *flow* to) data of type $\Box B$, thus all programs of type $\Box\mathsf{Bool} \Rightarrow \mathsf{Bool}$ must be noninterferent. To show this, we analyse the possible normal forms of f and observe that they must be equivalent to a constant function, such as $\lambda\, x.\, \mathsf{true}$ or $\lambda\, x.\, \mathsf{false}$, which evidently does not use its input argument x and is thus noninterferent.

In detail, normal forms of type $\Box\mathsf{Bool} \Rightarrow \mathsf{Bool}$ must have the shape $\lambda\, x.\, m$, for some normal form $\cdot, \Box\mathsf{Bool} \vdash_{\mathrm{NF}} m : \mathsf{Bool}$. If m is either true or false, then $\lambda\, x.\, m$ must be a constant function and we are done. Otherwise, it must be some normal form $\cdot, \Box\mathsf{Bool} \vdash_{\mathrm{NF}} \mathsf{ifte}\, n\, m_1\, m_2 : \mathsf{Bool}$ with a neutral $n : \mathsf{Bool}$ either in context $\cdot$ or in context $\cdot, \Box\mathsf{Bool}$. Such a neutral could either be of shape $\mathsf{unbox}\, n'$ or $\mathsf{app}\, n''\, m'$ for some neutrals n' and n''. However, this is impossible, since the context of the neutral $\mathsf{unbox}\, n'$ must contain a lock, and neither the context $\cdot$ nor the context $\cdot, \Box\,\mathsf{Bool}$ do. The existence of n'' can also be similarly dismissed by appealing to the definition of neutrals.

Discussion Observe that not all Fitch-style calculi are well-suited for interpreting the type $\Box A$ as a secret, because noninterference might not hold. In λ_{IS4}, the term $\lambda\, x.\, \mathsf{unbox}\, x : \Box A \Rightarrow A$ (axiom T) is well-typed but leaks the secret x, thus breaking noninterference. The validity of the interpretation of $\Box A$ as a secret depends on the calculus under consideration and the axioms it exhibits.

E.5.3. Partial Evaluation

Davies and Pfenning (1996) and Davies and Pfenning (2001) present a modal type system for staged computation based on IS4. In their system, the type $\Box A$

represents *code* of type A that is to be executed at a later stage, and the axioms of IS4 correspond to operations that manipulate code. The axiom $\mathrm{K} : \Box(A \Rightarrow B) \Rightarrow (\Box A \Rightarrow \Box B)$ corresponds to substituting code in code, $\mathrm{T} : \Box A \Rightarrow A$ to evaluating code, and $4 : \Box A \Rightarrow \Box\Box A$ to further delaying the execution of code to a subsequent stage. A desired property of this type system is that code must only depend on code, and thus the term $\lambda\, x : A.\, \mathsf{box}\, x$ must be ill-typed.

Although λ_{IS4} exhibits the desired properties of a type system for staging, its equational theory in Figure E.12 does not reflect the semantics of staged computation. For example, the result of normalizing the term $\mathsf{box}\,(2 * \mathsf{unbox}\,(\mathsf{box}\,3))$ in λ_{IS4} extended with natural number literals and multiplication is $\mathsf{box}\,6$. While the result expected from reducing it in accordance with Davies and Pfenning's operational semantics is $\mathsf{box}\,(2 * 3)$. The equational theory of Fitch-style calculi in general do not take into account the occurrence of a term (such as the literal 3) under box, while this is crucial for Davies and Pfenning's semantics. We return to this discussion at the end of this section.

If we restrict our attention to a special case of staged computation in partial evaluation (Jones et al. 1993), however, the semantics of Fitch-style calculi are better suited. In the context of partial evaluation, the type $\Box A$ represents a *dynamic* computation of type A that must be executed at runtime, and other types represent *static* computations. Static and dynamic are also known as binding-time annotations, and they are used by a partial evaluator to evaluate all static computations.

In the term $\mathsf{box}\,(2 * \mathsf{unbox}\,(\mathsf{box}\,3))$, we consider the literal 3 to be annotated as dynamic since it occurs under box. The construct unbox strips this annotation and brings it back to static. The multiplication of static subterms 2 and $\mathsf{unbox}\,(\mathsf{box}\,3)$ is however considered annotated dynamic since it itself occurs under box. As a result, a partial evaluator that respects these annotations does not perform the multiplication and specializes the term to $\mathsf{box}\,(2 * 3)$—which matches the result of evaluating with Davies and Pfenning's staging semantics. Observe that the same partial evaluator would specialize the expression $2 * \mathsf{unbox}\,(\mathsf{box}\,3)$ to 6 since the multiplication does not occur under box and is thus considered to be annotated static.

The goal of a partial evaluator is to optimize runtime execution of a program by eagerly evaluating as many static computations as possible and yielding an optimal dynamic program. The term $\mathsf{box}\,6$ is more optimized than the term $\mathsf{box}\,(2 * 3)$ since the evidently static multiplication has also been evaluated. Normalization in a Fitch-style calculus yields the former result, and the gain in optimality can be seen as a form of *binding-time improvement* (Jones et al. 1993) that is performed automatically during normalization.

In this section, we extend λ_{IK} with natural number literals and multiplication

(denoted λ_{IK}+Nat), and extend the NbE model of λ_{IK} to λ_{IK}+Nat. We use λ_{IK} as the base calculus since the other axioms are not needed in the context of partial evaluation (Davies and Pfenning 1996; Davies and Pfenning 2001). The resulting normalization function yields an optimal partial evaluator for λ_{IK}+Nat. In partial evaluation, as with staging in general, we desire that a term $\lambda\, x : \mathsf{N}.\, \mathsf{box}\, x$ be disallowed, since a runtime execution of a dynamic computation must not have a static dependency. While this term is already ill-typed in λ_{IK}+Nat, we prove a kind of binding-time correctness property for λ_{IK}+Nat that implies that *no* term equivalent to $\lambda\, x : \mathsf{N}.\, \mathsf{box}\, x$ can exist.

Extension with Natural Number Literals and Multiplication We extend λ_{IK} with a type N, a construct lift for including natural number literals, and an operation $*$ for multiplying terms of type N—as described in Figure E.17.

$$\text{Ty} \quad A, B ::= \ldots \mid \mathsf{N} \qquad\qquad \text{Ctx} \quad \Gamma ::= \ldots$$

$$\frac{}{\Gamma \vdash \mathsf{lift}\, k : \mathsf{N}}\ \textsc{N-Lift}\ k \in \mathbb{N} \qquad\qquad \frac{\Gamma \vdash t_1 : \mathsf{N} \qquad \Gamma \vdash t_2 : \mathsf{N}}{\Gamma \vdash t_1 * t_2 : \mathsf{N}}\ \textsc{N-Mul}$$

Figure E.17.: Types, contexts, intrinsically-typed terms of λ_{IK}+Nat (omitting the unchanged rules of Figure E.5)

We extend the equational theory of λ_{IK} with some rules such as $\mathsf{lift}\, k_1 * \mathsf{lift}\, k_2 \sim \mathsf{lift}\, (k_1 * k_2)$ (for natural numbers k_1 and k_2), $\mathsf{lift}\, 0 * t \sim \mathsf{lift}\, 0$, $t \sim \mathsf{lift}\, 1 * t$, $t * \mathsf{lift}\, k \sim \mathsf{lift}\, k * t$, etc. The normal forms of λ_{IK}+Nat include those of λ_{IK} in addition to the following.

$$\textsc{Nf}/\mathsf{N}_1\ \ \Gamma \vdash_{\text{NF}} \mathsf{lift}\, 0 : \mathsf{N} \qquad\qquad \frac{\Gamma \vdash_{\text{NE}} n_1 : \mathsf{N} \qquad \ldots \qquad \Gamma \vdash_{\text{NE}} n_j : \mathsf{N}}{\Gamma \vdash_{\text{NF}} \mathsf{lift}\, k * n_1 * \cdots * n_j : \mathsf{N}}\ \textsc{Nf}/\mathsf{N}_2\ k \in \mathbb{N} \setminus \{0\}$$

The normal form $\mathsf{lift}\, k * n_1 * \cdots * n_j$ denotes a multiplication of a nonzero literal with a sequence of neutrals of type N, which can possibly be empty. The term $\mathsf{box}\, (2 * \mathsf{unbox}\, (\mathsf{box}\, 3))$ from earlier can be represented in λ_{IK}+Nat as $\mathsf{box}\, (\mathsf{lift}\, 2 * \mathsf{unbox}\, (\mathsf{box}\, (\mathsf{lift}\, 3)))$, and its normal form as $\mathsf{box}\, (\mathsf{lift}\, 6)$. To extend the NbE model for λ_{IK} to natural number literals and multiplication, we use the interpretation presented by Valliappan, Russo, et al. (2021) for normalizing arithmetic expressions. Omitting the rule $\mathsf{lift}\, 0 * t \sim \mathsf{lift}\, 0$, this interpretation

also resembles the one constructed systematically in the framework of Yallop et al. (2018) for commutative monoids.

Proving Binding-Time Correctness Binding-time correctness for a term $\cdot \vdash f : \mathsf{N} \Rightarrow \Box\mathsf{N}$ can be stated similar to noninterference: it must be the case that $\cdot \vdash \mathsf{app}\, f\, u_1 \sim \mathsf{app}\, f\, u_2 : \Box\mathsf{N}$ for any two arguments $\cdot \vdash u_1, u_2 : \mathsf{N}$. The satisfaction of this property implies that no well-typed term equivalent to $\lambda\, x : \mathsf{N}.\, \mathsf{box}\, x$ exists, since applying it to different arguments would yield different results. As before with noninterference, we can prove this property by case analysis on the possible normal forms of f. A normal form of f is either of the form $\lambda\, x.\, \mathsf{box}\, (\mathsf{lift}\, 0)$ or $\lambda\, x.\, \mathsf{box}\, (\mathsf{lift}\, k * n_1 * \cdots * n_j)$ for some natural number k and neutrals $n_1, \ldots, n_j$ of type N in context $\cdot, \mathsf{N}, \blacksquare$. In the former case, we are done immediately since $\lambda\, x.\, \mathsf{box}\, (\mathsf{lift}\, 0)$ is a constant function that evidently satisfies the desired criterion. In the latter case, we observe by induction that no such neutrals n_i exist, and hence f must be equivalent to the function $\lambda\, x.\, \mathsf{box}\, (\mathsf{lift}\, k)$, which is also constant.

As a part of binding-time correctness, we may also desire that nonconstant terms $\Box A \Rightarrow A$ like $\lambda\, x : \Box A.\, \mathsf{unbox}\, x$ be disallowed since a static computation must not have a dynamic dependency. This can also be shown by following an argument similar to the proof of noninterference in Section E.5.2.

Discussion The operational semantics for staged computation is given by Davies and Pfenning via translation to a dual-context calculus for IS4, where evaluation under the introduction rule box for $\Box$ is disallowed. While it is possible to implement a normalization function for λ_{IS4} that does not normalize under box, this then misses certain reductions that *are* enabled by the translation. For instance, the term $\mathsf{box}\, (2 * \mathsf{unbox}\, (\mathsf{box}\, 3))$ is already in normal form if we simply disallow normalization under box, while the translation ensures the reduction of $\mathsf{unbox}\, (\mathsf{box}\, 3)$ by reducing the term to $\mathsf{box}\, (2 * 3)$. This mismatch, in addition to the lack of a model for their system, makes the applicability of Fitch-style calculi for staged computation unclear.

E.6. Related and Further Work

Fitch-Style Calculi Fitch-style modal type systems (Borghuis 1994; Martini and Masini 1996) adapt the proof methods of Fitch-style natural deduction systems for modal logic. In a Fitch-style natural deduction system, to eliminate a formula $\Box A$, we open a so-called strict subordinate proof and apply an "import" rule to produce a formula A. Fitch-style lambda calculi achieve a

similar effect, for example in λ_{IK}, by adding a 🔒 to the context. To introduce a formula $\Box A$, on the other hand, we close a strict subordinate proof, and apply an "export" rule to a formula A—which corresponds to removing a 🔒 from the context. In the possible-world reading, adding a 🔒 corresponds to travelling to a future world, and removing it corresponds to returning to the original world.

The Fitch-style calculus λ_{IK} was presented for the logic IK by Borghuis (1994) and Martini and Masini (1996), and later investigated further by Clouston (2018). Clouston showed that 🔒 can be interpreted as the left adjoint of $\Box$, and proves a completeness result for a term calculus that extends λ_{IK} with a type former ⧫ that internalizes 🔒. The extended term calculus is, however, somewhat unsatisfactory since the normal forms do not enjoy the subformula property. Normalization was also considered by Clouston, but only with Rule $\Box$-β and not Rule $\Box$-η. The normalization result presented here considers both rules, and the corresponding completeness result achieved using the NbE model does not require the extension of λ_{IK} with ⧫. The decidability result that follows for the complete equational theory of λ_{IK} also appears to have been an open problem prior to our work.

For the logic IS4, there appear to be several possible formulations of a Fitch-style calculus, where the difference has to do with the definition of the rule $\lambda_{\mathrm{IS4}}/\Box$-Elim. One possibility is to define unbox by explicitly recording the context extension as a part of the term former. Davies and Pfenning (1996) and Davies and Pfenning (2001) present such a system where they annotate the term former unbox as unbox_n to denote the number of 🔒s. Another possibility is to define unbox without any explicit annotations, thus leaving it ambiguous and to be inferred from a specific typing derivation. Such a system is presented by Clouston (2018), and also discussed by Davies and Pfenning. In either formulation terms of type $\Box A \Rightarrow A$ (axiom T) and $\Box A \Rightarrow \Box\Box A$ (axiom 4) that satisfy the comonad laws are derivable. As a result, both formulations exhibit the logical equivalence $\Box\Box A \Leftrightarrow \Box A$. The primary difference lies in whether this logical equivalence can also be shown to be an isomorphism, i.e. whether the semantics of the modality $\Box$ is a comonad which is also *idempotent*. In Clouston's categorical semantics the modality $\Box$ is interpreted by an idempotent comonad. The λ_{IS4} calculus presented here falls under the former category, where we record the extension explicitly using a premise instead of an annotation.

Gratzer, Sterling, et al. (2019) present yet another possibility that reformulates the system for IS4 in Clouston (2018). They further extend it with dependent types, and also prove a normalization result using NbE with respect to an equational theory that includes both Rule $\Box$-β and Rule $\Box$-η. Although their approach is semantic in the sense of using NbE, their semantic domain has a very syntactic flavour (Gratzer, Sterling, et al. 2019, Section 3.2) that

obscures the elegant possible-world interpretation. For example, it is unclear as to how their NbE algorithm can be adapted to minor variations in the syntax such as in λ_{IK}, λ_{IK4} and λ_{IT}—a solution to which is at the very core of our pursuit. This difference also has to do with the fact that they are interested in NbE for type-checking (also called "untyped" or "defunctionalized" NbE), while we are interested in NbE for well-typed terms (and thus "typed" NbE), which is better suited for studying the underlying models. Furthermore, we also avoid several complications that arise in accommodating dependent types in a Fitch-style calculus, which is the main goal of their work.

Davies and Pfenning present their calculus for IS4 using a stack of contexts, which they call "Kripke-style", as opposed to the single Fitch-style context with a first-class delimiting operator 🔒. The elimination rule unbox_n for $\Box$ in the Kripke-style calculus for IS4 is indexed by an arbitrary natural number n specifying the number of stack frames the rule adds to the context stack of its premise. This index n corresponds to the modal accessibility premise of the Fitch-style unbox rule presented in Figure E.11. As in the Fitch-style presentation, Kripke-style calculi corresponding to the other logics IK, IT and IK4 can be recovered by restricting the natural numbers n for which the unbox_n rule is available. Hu and Pientka (2022) present a normalization by evaluation proof for the Kripke-style calculi for all four logics IK, IT, IK4, and IS4. Their solution has a syntactic flavour similar to Gratzer, Sterling, et al. (2019) and also does not leverage the possible-world semantics. Furthermore, their proof is given for a single parametric system that encompasses the modal logics of interest, which need not be possible when we consider further modal axioms such as $\mathrm{R} : A \Rightarrow \Box A$.

Possible-World Semantics for Fitch-Style Calculi Given that Fitch-style natural deduction for modal logic has itself been motivated by possible-world semantics, it is only natural that Fitch-style calculi can also be given possible-world semantics. It appears to be roughly understood that the 🔒 operator models some notion of a past world, but this has not been—to the best of our knowledge—made precise with a concrete definition that is supported by a soundness and completeness result. As noted earlier, this requires a minor refinement of the frame conditions that define possible-world models for intuitionistic modal logic given by Božić and Došen (1984).

Dual-Context Calculi Dual-context calculi (Pfenning and Davies 2001; Davies and Pfenning 1996; Davies and Pfenning 2001; Kavvos 2020) provide an alternative approach to programming with the necessity modality using judgements

of the form $\Delta; \Gamma \vdash A$ where Δ is thought of as the modal context and Γ as the usual (or "local") one. As opposed to a "direct" eliminator as in Fitch-style calculi, dual-context calculi feature a pattern-matching eliminator formulated as a let-construct. The let-construct allows a type $\Box A$ to be eliminated into an arbitrary type C, which induces an array of commuting conversions in the equational theory to attain normal forms that obey the subformula property. Furthermore, the inclusion of an η-law for the $\Box$ type former complicates the ability to produce a unique normal form. Normalization (and, more specifically, NbE) for a pattern-matching eliminator—while certainly achievable—is a much more tedious endeavour, as evident from the work on normalizing sum types (Altenkirch, Dybjer, et al. 2001; Lindley 2007; Abel and Sattler 2019), which suffer from a similar problem. Presumably for this reason, none of the existing normalization results for dual-context calculi consider the η-law. The possible-world semantics of dual-context calculi is also less apparent, and it is unclear how NbE models can be constructed as instances of that semantics.

Multimodal Type Theory (MTT) Gratzer, Kavvos, et al. (2020) present a multimodal dependent type theory that for every choice of mode theory yields a dependent type theory with multiple interacting modalities. In contrast to Fitch-style calculi, their system features a variable rule that controls the use of variables of modal type in context. Further, the elimination rule for modal types is formulated in the style of the let-construct for dual-context calculi. In a recent result, Gratzer (2021) proves normalization for multimodal type theory. In spite of the generality of multimodal type theory, it is worth noting that the normalization problem for Fitch-style calculi, when considering the full equational theory, is not a special case of normalization for multimodal type theory.

Further Modal Axioms The possible-world semantics and NbE models presented here only consider the logics IK, IT, IK4 and IS4. We wonder if it would be possible to extend the ideas presented here to further modal axioms such as $\mathrm{R} : A \Rightarrow \Box A$ and $\mathrm{GL} : \Box(\Box A \Rightarrow A) \Rightarrow \Box A$, especially considering that the calculi may differ in more than just the elimination rule for the $\Box$ type.

Data Availability Statement

The Agda mechanization (Valliappan, Ruch, and Tomé Cortiñas 2022a) of the calculi λ_{IK} and λ_{IS4} and their normalization algorithms are available in the Zenodo repository.

Acknowledgements

We would like to thank Andreas Abel, Thierry Coquand, and Graham Leigh for their feedback on earlier versions of this work. We would also like to thank the anonymous referees of both the paper and the artifact for their valuable comments and helpful suggestions.

This work is supported by the SSF under the projects Octopi (Ref. RIT17-0023R) and WebSec (Ref. RIT17-0011).

Bibliography

Abadi, Martín et al. (1999). "A Core Calculus of Dependency". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, pp. 147–160. DOI: 10.1145/292540.292555. URL: https://doi.org/10.1145/292540.292555 (cit. on p. 217).

[SW Rel.] Abel, Andreas, Guillaume Allais, et al., *Agda 2* version 2.6.2.1, 2005–2021. Chalmers University of Technology and Gothenburg University. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php (cit. on p. 191).

Abel, Andreas and Christian Sattler (2019). "Normalization by Evaluation for Call-By-Push-Value and Polarized Lambda Calculus". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by Ekaterina Komendantskaya. ACM, 3:1–3:12. DOI: 10.1145/3354166.3354168. URL: https://doi.org/10.1145/3354166.3354168 (cit. on pp. 219, 225).

Ahman, Danel and Sam Staton (2013). "Normalization by Evaluation and Algebraic Effects". In: *Proceedings of the Twenty-ninth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2013, New Orleans, LA, USA, June 23-25, 2013*. Ed. by Dexter Kozen and Michael W. Mislove. Vol. 298. Electronic Notes in Theoretical Computer Science. Elsevier, pp. 51–69. DOI: 10.1016/j.entcs.2013.09.007. URL: https://doi.org/10.1016/j.entcs.2013.09.007 (cit. on p. 215).

Altenkirch, Thorsten, Peter Dybjer, et al. (2001). "Normalization by Evaluation for Typed Lambda Calculus with Coproducts". In: *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, pp. 303–310. DOI: 10.1109/LICS.2001.932506. URL: https://doi.org/10.1109/LICS.2001.932506 (cit. on p. 225).

Altenkirch, Thorsten and Tarmo Uustalu (2004). "Normalization by Evaluation for lambda^{-2}". In: *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings*. Ed. by Yukiyoshi Kameyama and Peter J. Stuckey. Vol. 2998. Lecture Notes in Computer Science. Springer, pp. 260–275. DOI: 10.1007/978-3-540-24754-8_19. URL: https://doi.org/10.1007/978-3-540-24754-8%5C_19 (cit. on p. 219).

[SW] Augustsson, Lennart et al., *Haskell* 1990. URL: https://www.haskell.org/ (cit. on p. 213).

Berger, Ulrich and Helmut Schwichtenberg (1991). "An Inverse of the Evaluation Functional for Typed lambda-calculus". In: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*. IEEE Computer Society, pp. 203–211. DOI: 10.1109/LICS.1991.151645. URL: https://doi.org/10.1109/LICS.1991.151645 (cit. on p. 190).

Borghuis, Valentijn Anton Johan (1994). *Coming to terms with modal logic*. On the interpretation of modalities in typed λ-calculus, Dissertation, Technische Universiteit Eindhoven, Eindhoven, 1994. Technische Universiteit Eindhoven, Eindhoven, pp. x+219 (cit. on pp. 189, 222, 223).

Božić, Milan and Kosta Došen (1984). "Models for normal intuitionistic modal logics". In: *Studia Logica* 43.3, pp. 217–245. ISSN: 0039-3215. DOI: 10.1007/BF02429840. URL: https://doi.org/10.1007/BF02429840 (cit. on pp. 192, 193, 224).

Choudhury, Vikraman and Neel Krishnaswami (2020). "Recovering purity with comonads and capabilities". In: *Proc. ACM Program. Lang.* 4.ICFP, 111:1–111:28. DOI: 10.1145/3408993. URL: https://doi.org/10.1145/3408993 (cit. on pp. 189, 212–214).

Clouston, Ranald (2018). "Fitch-Style Modal Lambda Calculi". In: *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Christel Baier and Ugo Dal Lago. Vol. 10803. Lecture Notes in Computer Science. Springer, pp. 258–275. DOI: 10.1007/978-3-319-89366-2_14. URL: https://doi.org/10.1007/978-3-319-89366-2%5C_14 (cit. on pp. 189, 190, 193, 196, 201, 206, 209, 211, 218, 223).

Coquand, Catarina (2002). "A Formalised Proof of the Soundness and Completeness of a Simply Typed Lambda-Calculus with Explicit Substitutions". In:

High. Order Symb. Comput. 15.1, pp. 57–90. DOI: 10.1023/A:1019964114625. URL: https://doi.org/10.1023/A:1019964114625 (cit. on p. 194).

Davies, Rowan and Frank Pfenning (1996). "A Modal Analysis of Staged Computation". In: *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. Ed. by Hans-Juergen Boehm and Guy L. Steele Jr. ACM Press, pp. 258–270. DOI: 10.1145/237721.237788. URL: https://doi.org/10.1145/237721.237788 (cit. on pp. 189, 219, 221, 223, 224).

– (2001). "A modal analysis of staged computation". In: *J. ACM* 48.3, pp. 555–604. DOI: 10.1145/382780.382785. URL: https://doi.org/10.1145/382780.382785 (cit. on pp. 189, 219–224).

Ewald, W. B. (1986). "Intuitionistic Tense and Modal Logic". In: *J. Symb. Log.* 51.1, pp. 166–179. DOI: 10.2307/2273953. URL: https://doi.org/10.2307/2273953 (cit. on p. 192).

Filinski, Andrzej (2001). "Normalization by Evaluation for the Computational Lambda-Calculus". In: *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings*. Ed. by Samson Abramsky. Vol. 2044. Lecture Notes in Computer Science. Springer, pp. 151–165. DOI: 10.1007/3-540-45413-6_15. URL: https://doi.org/10.1007/3-540-45413-6%5C_15 (cit. on p. 215).

Fischer-Servi, Gisèle (1981). "Semantics for a class of intuitionistic modal calculi". In: *Italian studies in the philosophy of science*. Vol. 47. Boston Stud. Philos. Sci. Reidel, Dordrecht-Boston, Mass., pp. 59–72 (cit. on p. 192).

[SW] Freeman, Phil, *PureScript* 2013. URL: https://www.purescript.org/ (cit. on p. 213).

Goguen, Joseph A. and José Meseguer (1982). "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, pp. 11–20. DOI: 10.1109/SP.1982.10014. URL: https://doi.org/10.1109/SP.1982.10014 (cit. on p. 217).

Gratzer, Daniel (2021). "Normalization for multimodal type theory". In: *CoRR* abs/2106.01414. arXiv: 2106.01414. URL: https://arxiv.org/abs/2106.01414 (cit. on p. 225).

Gratzer, Daniel, G. A. Kavvos, et al. (2020). "Multimodal Dependent Type Theory". In: *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*. Ed. by Holger Hermanns et al. ACM, pp. 492–506. DOI: 10.1145/3373718.3394736. URL: https://doi.org/10.1145/3373718.3394736 (cit. on p. 225).

Gratzer, Daniel, Jonathan Sterling, et al. (2019). "Implementing a modal dependent type theory". In: *Proc. ACM Program. Lang.* 3.ICFP, 107:1–107:29. DOI: 10.1145/3341711. URL: https://doi.org/10.1145/3341711 (cit. on pp. 223, 224).

Hu, Jason Z. S. and Brigitte Pientka (2022). "An Investigation of Kripke-style Modal Type Theories". In: *CoRR* abs/2206.07823. arXiv: 2206.07823. URL: https://arxiv.org/abs/2206.07823 (cit. on p. 224).

Jay, C. Barry and Neil Ghani (1995). "The Virtues of Eta-Expansion". In: *J. Funct. Program.* 5.2, pp. 135–154. DOI: 10.1017/S0956796800001301. URL: https://doi.org/10.1017/S0956796800001301 (cit. on p. 190).

Jones, Neil D. et al. (1993). *Partial evaluation and automatic program generation.* Prentice Hall international series in computer science. Prentice Hall. ISBN: 978-0-13-020249-9 (cit. on p. 220).

Kavvos, G. A. (2020). "Dual-Context Calculi for Modal Logic". In: *Log. Methods Comput. Sci.* 16.3. DOI: 10.23638/LMCS-16(3:10)2020. URL: https://lmcs.episciences.org/6722 (cit. on pp. 213, 224).

Lindley, Sam (2007). "Extensional Rewriting with Sums". In: *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings.* Ed. by Simona Ronchi Della Rocca. Vol. 4583. Lecture Notes in Computer Science. Springer, pp. 255–271. DOI: 10.1007/978-3-540-73228-0_19. URL: https://doi.org/10.1007/978-3-540-73228-0%5C_19 (cit. on p. 225).

Martini, Simone and Andrea Masini (1996). "A computational interpretation of modal proofs". In: *Proof theory of modal logic (Hamburg, 1993).* Vol. 2. Appl. Log. Ser. Kluwer Acad. Publ., Dordrecht, pp. 213–241. DOI: 10.1007/978-94-017-2798-3_12. URL: https://doi.org/10.1007/978-94-017-2798-3_12 (cit. on pp. 189, 222, 223).

Milner, Robin et al. (1990). *Definition of standard ML.* MIT Press. ISBN: 978-0-262-63132-7 (cit. on p. 212).

Miyamoto, Kenji and Atsushi Igarashi (2004). "A modal foundation for secure information flow". In: *In Proceedings of IEEE Foundations of Computer Security (FCS)*, pp. 187–203 (cit. on p. 189).

Moggi, Eugenio (1989). "Computational Lambda-Calculus and Monads". In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989.* IEEE Computer Society, pp. 14–23. DOI: 10.1109/LICS.1989.39155. URL: https://doi.org/10.1109/LICS.1989.39155 (cit. on p. 212).

– (1991). "Notions of Computation and Monads". In: *Inf. Comput.* 93.1, pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4. URL: https://doi.org/10.1016/0890-5401(91)90052-4 (cit. on pp. 214, 217).

Pfenning, Frank and Rowan Davies (2001). "A judgmental reconstruction of modal logic". In: *Math. Struct. Comput. Sci.* 11.4, pp. 511–540. DOI: 10.1017/S0960129501003322. URL: https://doi.org/10.1017/S0960129501003322 (cit. on pp. 213, 224).

Plotkin, Gordon D. and Colin Stirling (1986). "A Framework for Intuitionistic Modal Logics". In: *Proceedings of the 1st Conference on Theoretical Aspects of Reasoning about Knowledge, Monterey, CA, USA, March 1986*. Ed. by Joseph Y. Halpern. Morgan Kaufmann, pp. 399–406 (cit. on p. 192).

Russo, Alejandro et al. (2008). "A library for light-weight information-flow security in haskell". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, pp. 13–24. DOI: 10.1145/1411286.1411289. URL: https://doi.org/10.1145/1411286.1411289 (cit. on p. 217).

Sabelfeld, Andrei and Andrew C. Myers (2003). "Language-based information-flow security". In: *IEEE J. Sel. Areas Commun.* 21.1, pp. 5–19. DOI: 10.1109/JSAC.2002.806121. URL: https://doi.org/10.1109/JSAC.2002.806121 (cit. on p. 216).

Shikuma, Naokata and Atsushi Igarashi (2008). "Proving Noninterference by a Fully Complete Translation to the Simply Typed Lambda-Calculus". In: *Log. Methods Comput. Sci.* 4.3. DOI: 10.2168/LMCS-4(3:10)2008. URL: https://doi.org/10.2168/LMCS-4(3:10)2008 (cit. on p. 217).

Simpson, Alex K. (1994). "The proof theory and semantics of intuitionistic modal logic". PhD thesis. University of Edinburgh, UK. URL: https://hdl.handle.net/1842/407 (cit. on pp. 192, 193).

Tomé Cortiñas, Carlos and Nachiappan Valliappan (2019). "Simple Noninterference by Normalization". In: *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, CCS 2019, London, United Kingdom, November 11-15, 2019*. Ed. by Piotr Mardziel and Niki Vazou. ACM, pp. 61–72. DOI: 10.1145/3338504.3357342. URL: https://doi.org/10.1145/3338504.3357342 (cit. on p. 217).

[SW Rel.] Valliappan, Nachiappan, Fabian Ruch, and Carlos Tomé Cortiñas, *Artifact for "Normalization for Fitch-Style Modal Calculi"* version 1.1.0, Aug. 2022. DOI: 10.5281/zenodo.6957191, URL: https://doi.org/10.5281/zenodo.6957191 (cit. on pp. 191, 225).

Valliappan, Nachiappan, Fabian Ruch, and Carlos Tomé Cortiñas (2022b). "Normalization for fitch-style modal calculi". In: *Proc. ACM Program. Lang.* 6.ICFP, pp. 772–798. DOI: 10.1145/3547649. URL: https://doi.org/10.1145/3547649 (cit. on p. 187).

Valliappan, Nachiappan, Alejandro Russo, et al. (2021). "Practical normalization by evaluation for EDSLs". In: *Haskell 2021: Proceedings of the 14th ACM*

SIGPLAN International Symposium on Haskell, Virtual Event, Korea, August 26-27, 2021. Ed. by Jurriaan Hage. ACM, pp. 56–70. DOI: 10.1145/3471874.3472983. URL: https://doi.org/10.1145/3471874.3472983 (cit. on pp. 215, 221).

Yallop, Jeremy et al. (2018). "Partially-static data as free extension of algebras". In: *Proc. ACM Program. Lang.* 2.ICFP, 100:1–100:30. DOI: 10.1145/3236795. URL: https://doi.org/10.1145/3236795 (cit. on p. 222).