SEMANTICS FOR SECURE SOFTWARE

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Andrew Karl Hirsch

August 2019

SEMANTICS FOR SECURE SOFTWARE

Andrew Karl Hirsch, Ph.D.

Cornell University 2019

In order to build machine-checked proven-secure software, we need formal security policies that express what it means to be "secure." We must then show that the semantics of our software matches the semantics of those policies. This requires formal semantics for both programs and policies. In this dissertation, we explore the semantics of effectful programs and the semantics of authorization policies.

The most well-known class of effects are those that can be given semantics via a monad, though current research also focuses on those that can be given a semantics via a comonad. We compare three methods for combining these two popular options: one method requires extra semantic structure, whereas the other methods can be applied to any monadic and comonadic effects. If the extra semantic structure needed for the first method exists then the three semantics are equivalent. Otherwise, we show that the two remaining semantics correspond to strict and lazy interpretations of the effects.

On the other side, we use authorization logics to express authorization policies. Authorization logics can be given semantics using either models or a proof system. We build a model theory for an authorization logic that more-closely expresses how authorization logics are used by systems than traditional models. We also build a proof system for an authorization logic that ensures that proofs of authorization respect information-security policies.

BIOGRAPHICAL SKETCH

Andrew K. Hirsch was born and raised in Fort Worth, Texas. In 2009, he left for undergraduate work at The George Washington University in Washington, D.C., where he earned a Bachelor's of Science degree in computer science and pure mathematics. In 2013, he joined the computer science department of Cornell University in Ithaca, New York.

To my parents, who raised me to be curious and not a bit stubborn.

To my partner, who is my rock.

ACKNOWLEDGEMENTS

There are many people to acknowledge who have helped me get to this point. In particular, I want to thank my advisor, Ross Tate, and my collaborators Pedro H. Azevedo de Amorim, Owen Arden, Ethan Cecchetti, and Michael Clarkson. I also want to thank my mentors at The George Washington University: Valentina Harizanov, Bagchi Narahari, Gabe Parmer, Rahul Simha, and Poorvi Vora, as well as those at Cornell University: Kavita Bala, Robert Kleinberg, Dexter Kozen, Andrew Myers, Adrian Sampson, Fred Schneider, and Emin Gün Sirer.

Many friends and colleagues have helped me through my time in graduate school. Besides those above, I especially want to thank Shrutarshi Basu, Soumya Basu, Eleanor Birrell, Eric Campbell, Natacha Crooks, Jonathan Dilorenzo, Molly Feldman, Dietrich Geisler, Jed Liu, Tom Magrino, Matthew Milano, Andrew Morgan, Fabian Mühlböck, Vlad Nicolae, Rolf Recto, Oliver Richardson, Michael Roberts, Xanda Schofield, Eston Schweickart, Isaac Sheff, and Drew Zagieboylo. They have been my support group both emotionally and professionally.

I will also take the chance to thank those who made a difference in the papers which this dissertation is based around. Stephen Brooks, Marco Gaboardi, and Guillaume Much-Maccognoni all provided good discussions and feedback regarding layering, and the Cornell Programming Languages Discussion Group (of which many members are listed here) gave feedback and helped immensely with editing. Martín Abadi, Adam Chlipala, Deepak Garg, Joe Halpern, Kristopher Micinski, Dexter Kozen, Fred Schneider, Colin Stirling, and Kevin Walsh all had discussions and comments that contributed to the chapter on belief semantics, and the coq-club mailing list was very helpful when developing the code. The work on belief semantics was supported in part by AFOSR grants F9550-06-0019, FA9550-11-1-0137, and FA9550-12-1-0334, NSF grants 0430161, 0964409, and CCF-0424422

(TRUST), ONR grants N00014-01-1-0968 and N00014-09-1-0652, and a grant from Microsoft. Coşku Acay, Arthur Azevedo de Amorim, Eric Campbell, Dietrich Geisler, Elisavet Kozyri, Jed Liu, Tom Magrino, Matthew Milano, Andrew Morgan, Andrew Myers, and Drew Zagieboylo gave invaluable feedback on FLAFOL. Funding for FLAFOL was provided in part by NSF grant #1704788, NSF CAREER grant #1750060, and a National Defense Science and Engineering Graduate Fellowship.

Finally, I want to take a chance to thank all of those who I have not named so far. Many have labored anonymously, such as reviewers who have helped me become a much better writer and researcher. I appreciate all of your work immensely.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Software manipulates most data. We keep most of our financial records in cyberspace, socialize via social media, and even perform mundane tasks like university registration online. This software must provide results while protecting our privacy and ensuring that others cannot harm us. In order to provide such assurances, we need machine-checked, proven-secure software, which requires a semantic understanding of both effectful programs and security policies.

Imagine a college student named Alice, who interacts with many digital systems. She wants to go online to look at her grades and sign up for classes while being sure that her mother cannot do so on her behalf. She shares pictures of college life with her family on Facebook, but she wants to prevent her father from immediately sharing them all with everyone he knows as well. Finally, she wants assurance that her credit card will incur no charges without her permission. These are all examples of *security policies* that Alice wants the systems that she interacts with to follow.

Poorly-designed policies or policies that interact in unexpected ways can have unintended consequences and so harm more than they help. For instance, imagine that Alice's university has a policy that parents or other students may not affect or view a student's registrar account, including seeing that student's grades. As a consequence either (a) Alice's mother cannot pay for Alice's tuition (since that would affect Alice's registrar account), or (b) Alice must share her registrar account with her mother, rendering Alice's policy that Alice's mother cannot see her grades violable. The university can get out of this dilemma by adding subaccounts which may pay a particular student's tuition bill but not see their grades. In order

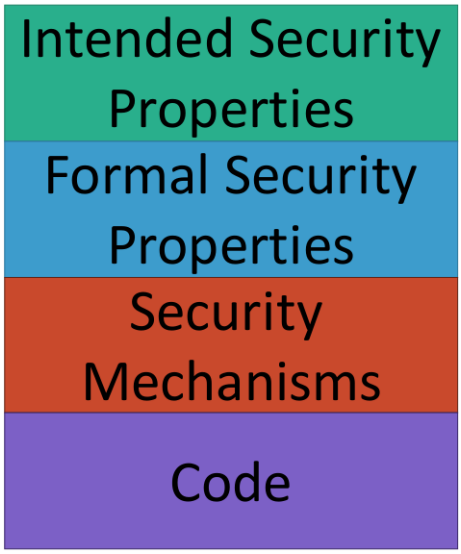
to ensure that no further such dilemmas appear, the university may wish to mathematically prove that their policies are coherent. However, so far we've articulated only *informal* policies, which we cannot possibly prove anything about. In order to provide assurances about security policies, we must create mathematically-precise *formal* policies, which are amenable to proof. For example, the university may write its policies in a formal logic, and then prove a (meta-)theorem stating that subaccounts may pay the tuition for their main account, but they may not see the grades associated with that account.

When we design systems, we design algorithms and data structures called *security mechanisms* (often shortened to just *mechanisms*) to help us enforce security policies. For instance, when Alice logs into her university registrar website the website might install a cookie on her browser, identifying her to other pages of the site. That cookie contains a timestamp which the site uses to ensure that it is still Alice using the account. Thus when Alice tries to check her grades, the site will check that timestamp and, if it has been too long since Alice logged in, invalidate the cookie so that Alice is required to log in again. It will also ensure that the account Alice has logged into is her main account, rather than the subaccount her mother uses to pay tuition.

Note that in this example the registrar website needs to read and write state in Alice's browser, and it also needs to know the current time when setting the timestamp in Alice's cookie and when checking for a timeout. Reading and writing state is a classic example of an *effect*. Intuitively, we can think of effects as ways that programs "do more than they say on the tin," in the sense that they represent ways that programs can take actions which are not apparent in the program's type. Mechanisms often use some effects, such as reading and writing the state of a user's

browser, in order to achieve results. The code that implements these mechanisms might contain even more effects than the mechanism designer assumed. For instance, the code that sets a cookie in Alice's browser might throw an exception if Alice's browser rejects the cookie. A system designer who is trying to verify this code must reason about how this new effect interacts with the mechanism.

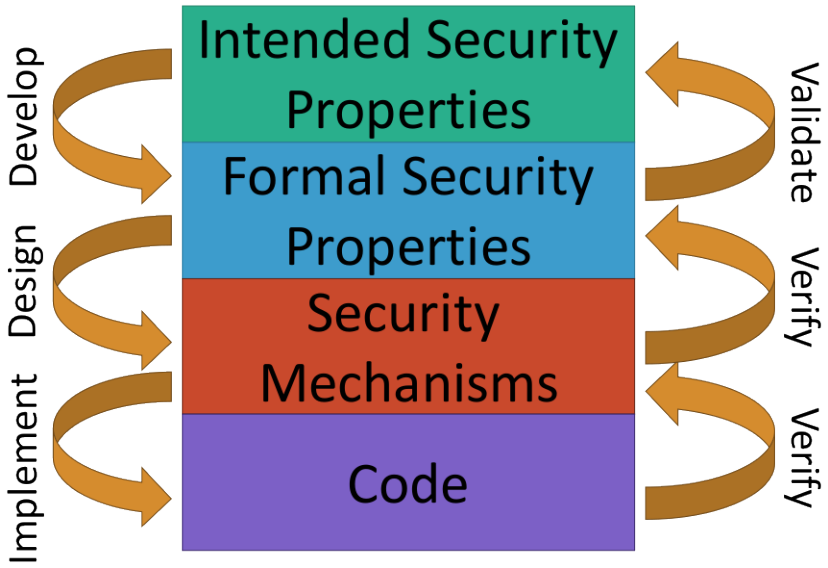
Ideally developers of secure software use the following stack of abstractions:



To use these abstractions, they start with informal security policies and use those to develop formal security policies. They use those formal security properties to develop security mechanisms, and then implement those mechanisms using effectful code. This process makes it easier to trust both the mechanisms themselves and the code that implements them.

In reality, developers often do not follow the above process, since they do not use formal security policies at all. This not only makes it more difficult to trust the code, but makes formal *verification* of security near impossible. To see why, consider the process used to formally verify security. First, we validate that our formal security policies reflect the informal policies that we started with and we

verify that our mechanisms correctly enforce our formal security policies. Finally, we verify that our code implements the mechanisms correctly. All together, we can visualize the development and verification process as:



We need to understand the semantics of both policies and code in order to partake in this process. In particular, verifying that code implements a security mechanism requires knowledge of the semantics of effectful code, while validation of a security mechanisms requires knowledge of the semantics of security policies. Verifying that a mechanism enforces a policy requires both. Let us examine each step in detail.

Verifying Code In order to verify that some code correctly implements a security mechanism, we must have an account of the semantics of effectful programs, since the mechanism uses some effects. The code might have more effects than the mechanism uses, and so we must understand how effects combine in order to establish the security properties of the code.

Semantics of Effectful Programs There are many ways to present program semantics. The security literature mostly presents *operational semantics*, which gives programs meaning via rules describing how they act on a computer. This can give extremely valuable insights by providing a concrete view of how programs operate. However, in this dissertation we use *denotational* semantics, especially *categorical semantics*. That is, we will interpret programs using mathematical structures called *categories*, giving us a more abstract view of programs and programming languages and allowing us to reason formally about commonalities between languages. Moreover, categorical semantics connects to related work on the semantics of effects, particularly through the use of *monads* [Moggi, 1989] and *comonads* [Petricek et al., 2012, 2014; Uustalu and Vene, 2005]. In Chapter 2, we will work on the boundary of denotational and operational semantics, using the categorical semantics of effects to give a denotational account of strictness and laziness.

Verifying Mechanisms In order to verify that a mechanism correctly enforces a security policy, we must have an account of the semantics of both effectful programs and security policies, since we must ensure that the semantics of the mechanism matches the semantics of the policy. Consider again the example where Alice's university registrar checks a cookie when she tries to view her grades on their website. They have an informal policy that only the student, and not their parents, can access the student's grades. They have a mechanism that attempts to enforce that informal policy by checking a cookie. In order to verify this mechanism, they have to know that the cookie check actually corresponds to the type of account, and we must know that the formal policy states that student accounts are allowed to view grades, but subaccounts are not.

Validating Formal Policies In order to validate that formal policies correspond to their informal counterpart, we must have an account of the semantics of security policies, since we must thoroughly understand the consequences of the formal policy. It is especially important here to understand how security policies combine, since this is where unintended consequences can rear their ugly head. For instance, in the above example the policy that parents not be able to affect their child's account conflicts with the policy that parents should be able to pay their child's bill. In order to see this conflict we must understand what these policies mean in context. In Chapter 4, we will see how we can reason about two types of policy together to prevent this kind of collision from happening between the two types of policy.

Semantics of Policies In this dissertation, we focus our discussion of security policies on *authorization* policies, which limit who may do what in a system. This is opposed to *authentication* policies, which discuss how parts of the system gain their roles, and *audit* policies, which ensure that actions in a system are logged properly. For instance, "subaccounts may not view student grades" is an authorization policy while "a student must have signed in within the last 5 minutes" is an authentication policy, and "a student's account name must be logged whenever that student looks at their grades" is an audit policy. Many policies are authorization policies, so our focus on authorization policies does not overly limit the results in this dissertation. In Chapter 4, we will also look at *information security* policies, which describe how data may be used.

Our main tool for understanding authorization policies is *authorization logic*. We define authorization logics as multi-modal logics with a notion of communication and a notion of trust, though this definition is far from universally accepted.

In other words, authorization logics have a *modality* (unary logical connective) for every *principal* (component of the system) symbolizing when that principal believes a formula to be true. Principals can then communicate their beliefs, but are only willing to communicate with principals they trust.

In this dissertation, we use two ways of giving semantics to logics: models and proof systems. Logicians define models to represent, as mathematical structures, the universes in which a logic describes truth. They then give meaning to a logical sentence by describing when that sentence is true in one of these universes. For instance, we can give meaning to a logic for reasoning about groups by describing when a formula is true about a group. Logicians usually interpret modal logics as reasoning about what *possible world* the reasoner can be in, which they formalize using a *Kripke* structure. Since Kripke structures do not have a way to talk about communication or trust, the few works on model theory for authorization logics [Genovese et al., 2012; Goguen and Meseguer, 1982] use modified versions of Kripke structures. However, these modified Kripke structures do not accurately reflect real systems, so it is unclear to what extent these models tell us when an authorization logic sentence is true about a system.

In Chapter 3, we develop *belief structures*, models which more-accurately reflect the design of real-world distributed systems. These models keep track of the *worldview* of each principal, where the worldview of p is defined as the set of formulae which that p believes. We then give a proof system for our logic, and show that our semantics is *sound*; that is, that every formula that can be proven in our proof system is true in every model. Furthermore, any Kripke structure induces a belief structure, showing that belief structures are at least as expressive as more-traditional Kripke structures.

In Chapter 4, we focus on *flow-limited authorization* [Arden and Myers, 2016; Arden et al., 2015], which combines reasoning about authorization with reasoning about information security. Information-security policies express how information may be used. For instance, an information-security policy might express that student grades are high-integrity, and thus that student accounts and subaccounts should not be able to affect grades. However, enforcing authorization policies can cause information-security policy violations. We give a first-order authorization logic for flow-limited authorization and discuss its proof theory, including a guarantee that information-security policies are respected by authorization proofs.

Organization To sum up, the rest of this dissertation is organized as follows:

- Chapter 2 gives a denotational account of strictness and laziness via effects. This also requires combining a producer effect with a consumer effect, an active area of research in the theory of effects. This chapter is based on joint work with my advisor, Ross Tate [Hirsch and Tate, 2018].
- Chapter 3 defines belief structures, which more-closely reflect how authorization logics are used in distributed systems. Belief semantics is not only sound for FOCAL and FOCALE—our example authorization logics—but that it is at least as expressive as the more-common Kripke semantics. This chapter is based on joint work with Michael Clarkson, done while we were both at The George Washington University [Hirsch and Clarkson, 2013a].
- Chapter 4 defines FLAFOL, a first-order logic for flow-limited authorization, and presents proof-theoretic results for FLAFOL. These proof-theoretic results make it easier to compare flow-limited authorization with previous work on authorization logic compared to FLAM, FLAFOL's immediate predeces-

sor. Moreover, we present a *non-interference* result, a powerful security result for authorization logics. This chapter is based on work with Pedro H. Azevedo de Amorim, Ethan Cecchetti, Ross Tate, and Owen Arden [Hirsch et al., 2019].

- In Chapter 5, we discuss ongoing and future work extending the semantics of both effectful programs and authorization policies, with a focus on flow-limited authorization. Moreover, we discuss how the two might be brought into direct contact; for instance, we describe ongoing work that reasons precisely about how effects and information-security policies interact.
- In Chapter 6, we discuss related work, including work done in monadic and comonadic semantics of effects, authorization logic, linear logic, and more.
- In Chapter 7, we conclude by discussing themes that run throughout the work presented in this dissertation.
- In Appendix A, we present the metatheory of a language used in Chapter 2 to understand how two effects interact. This includes proofs of progress and preservation for the type system of a language, along with a proof of termination and confluence for the operational semantics.
- In Appendix B, we give the full proof system for the logic in Chapter 4.
- Finally, in Appendix C, we give the full rules for *Compatible Supercontexts*, a proof-theoretic tool that we describe in Chapter 4.

CHAPTER 2

STRICT AND LAZY SEMANTICS FOR EFFECTS

The study of the semantics of effects has been quite fruitful. Researchers have found two particularly important kinds of effects: those that can be given semantics using a monad [Lucassen and Gifford, 1988; Marino and Milstein, 2009; Moggi, 1989; Nielson, 1996; Nielson and Nielson, 1999; Wadler and Thiemann, 1998], and those that can be given semantics using a comonad [Brookes and Geva, 1992; Brunel et al., 2014; Petricek et al., 2012, 2014; Uustalu and Vene, 2005, 2008].

Giving semantics to programs with both kinds of effects is an active area of research. The most common technique uses a technical device called a distributive law to describe the interaction of the effects when two effectful programs are composed. However, not every comonad-monad pair has a distributive law [Brookes and van Stone, 1993; Power and Watanabe, 2002], though often there is one. But importantly, while Brookes and van Stone had found several comonad-monad pairs without distributive laws that were interesting in the study of domain theory [Brookes and van Stone, 1993], they did not find any that corresponded to effects. Consequently, when Gaboardi et al. [2016] studied the semantics of languages with both kinds of effects, they decided to focus on the common case where a distributive law exists.

Here we give an example of a pair of effects with a monad and a comonad that do not have a distributive law. Furthermore, we demonstrate that *not* having a distributive law increases the forms of interaction these effects can have, with the difference between strictness and laziness arising as examples of interactions between these effects. This poses a challenge, though, because a distributive law is often used in order to compose such effectful programs.

Luckily, we do not need to come up with a completely new semantic technique to give semantics to programs with the pair of effects that make up our example. There are already two different semantic techniques that do not require a distributive law [Brookes and van Stone, 1993; Power and Watanabe, 2002]. These techniques previously went unnamed because they were believed to be unnecessary. Now that we have proven them to be useful, we dub them the *layerings*, one of which is the *monad-prioritizing* layering and the other of which is the *comonad-prioritizing* layering.

When it is possible to use a distributive law, all three semantic techniques necessarily produce equivalent results [Power and Watanabe, 2002]. However, since there is no distributive law in our example, the semantics specified by the layerings may produce different results. We show that, in fact, one of the layerings gives a *strict* semantics to programs, and the other gives a *lazy* semantics to programs.

This gives a new perspective on strictness and laziness, one of the oldest subjects in our field [Church and Rosser, 1936]. Instead of thinking of features like function application as being either strict or lazy, we think of strictness and laziness as arising from different interpretations of program composition. Of course there are intimate relationships between these perspectives, and hints of our perspective can be seen within many of the prior works on strictness and laziness. But we are bringing it to the forefront, making it bold and clear by showing how changing the way that programs compose can make even a language with almost no features—no functions, branches, or even arithmetic, and only one type ($\mathbb{N}$)—change between strict and lazy semantics.

Beyond giving an important example where the currently-preferred semantic technique based on distributive laws fails, our example has led us to develop a

classification of when that technique can and cannot apply. This classification describes the linguistic characteristics that the technique based on distributive laws requires, and contrasts these with the requirements of the layerings.

The rest of this chapter proceeds as follows:

- Section 2.1 introduces Comp, which is a minimal language with observationally distinct strict and lazy operational semantics.
- Section 2.2 brings attention to the effects in Comp that give rise to strictness and laziness, which we call consumer choice and producer choice. These will be the example effects for which semantics based on distributive laws do not work.
- Section 2.3 describes a comonad and a monad that capture consumer choice and producer choice, both of which come from classical linear logic. To assist in our exploration, we present several rules of classical and linear logic. The classical-logic rules here correspond to an effectful calculus (with consumer choice and producer choice), while the linear-logic rules here correspond to a (pure) calculus that makes the effects of classical logic explicit.
- Section 2.4 explores the three known techniques for giving semantics to a program with both kinds of effects, and gives a novel classification of the linguistic structure that each requires. We present the rules that formalize what we mean when we say a language is effectful, meaning rules that must be *admissable* in a language for that language to be considered effectful. We also refine these into rules for producer-effectful, consumer-effectful, and doubly-effectful (i.e. both producer-effectful and consumer-effectful) languages.
- Section 2.6 applies the two layerings to give categorical semantics to Comp. We show that the choice of layering reflects the choice between strict and

Variables	$x, y, z, \ldots$		
Constants	c	$::=$	$0 \mid 1 \mid \ldots$
Expressions	e	$::=$	$c \mid x \mid \mathbf{error}$
Statements	s	$::=$	$x \coloneqq e$
Programs	p	$::=$	$\cdot \mid p^+$
Non-Empty Programs	p^+	$::=$	$s_1; \ldots; s_n$
Types	t	$::=$	$\mathbb{N}$
Contexts	Γ	$::=$	$x : t, \ldots$
			(no repeats)
			(unordered)

Figure 2.1: Comp Syntax

lazy semantics. To do so, we introduce Proc, a calculus that captures the effects of consumer choice and producer choice in Comp using the comonad and monad from linear logic we explored in Section 2.3.

2.1 A Simple Language for Exploring Strictness and Laziness

We begin by presenting a language, called Comp, that we designed for exploring strictness and laziness. As one can see from the syntax in Figure 2.1, Comp is very simple. We designed it to have only those features that we need for our exploration.

The only possible statements in Comp are assignments, and a program is merely a list of assignments. For further simplicity, we assume that every variable is assigned at most one expression in a Comp program. The only complexity in Comp comes from the expressions that can be assigned to variables. First, constants can be assigned to variables. Second, one variable can be assigned to another. Finally, variables can be assigned **error**, which should be thought of as an expression that

$$\frac{}{\vdash x := c \dashv x : \mathbb{N}}\ \textsc{Const} \qquad \frac{}{y : t \vdash x := y \dashv x : t}\ \textsc{Var}$$

$$\frac{}{\vdash x := \mathbf{error} \dashv x : t}\ \textsc{Error} \qquad \frac{\Gamma \vdash p_1 \dashv x : t \qquad \Gamma', x : t \vdash p_2 \dashv y : t'}{\Gamma, \Gamma' \vdash p_1; p_2 \dashv y : t'}\ \textsc{Seq}$$

$$\frac{\Gamma \vdash p \dashv y : t'}{\Gamma, x : t \vdash p \dashv y : t'}\ \begin{matrix}(\textsc{Left}) \\ \textsc{Weakening}\end{matrix} \qquad \frac{\Gamma, x_1 : t, x_2 : t \vdash p \dashv y : t'}{\Gamma, x : t \vdash p[x_1, x_2 \mapsto x] \dashv y : t'}\ \begin{matrix}(\textsc{Left}) \\ \textsc{Contraction}\end{matrix}$$

Figure 2.2: Comp Typing Rules

throws an error when it is evaluated.

The output of a Comp program is considered to be the value assigned to the last variable in the program. For example, in the program $x := 3; y := 4$, the output is 4.

We provide a type system for Comp in Figure 2.2. The judgments used in the type system have the form $\Gamma \vdash p \dashv x : t$, where Γ is a set of variable-type pairs. We read this as "given inputs with types described by Γ, the output of p will be a value of type t assigned to x." This makes Comp a multiple-input, single-output language.

Note that the only possible type is $\mathbb{N}$ due to the simplistic nature of Comp. We present the type system here because the typing derivations are illuminating, though the types themselves are not. For instance, consider a derivation of $\vdash x := 3; y := \mathbf{error} \dashv y : \mathbb{N}$. As part of this we will need $x : \mathbb{N} \vdash y := \mathbf{error} \dashv y : \mathbb{N}$, which we cannot obtain by directly applying the Error rule because the Error rule requires an empty context. So instead, we must first apply another rule to get

the following derivation:

$$\dfrac{\dfrac{}{\vdash y \coloneqq \mathbf{error} \dashv y : \mathbb{N}}}{x : \mathbb{N} \vdash y \coloneqq \mathbf{error} \dashv y : \mathbb{N}}$$

This other rule discards the unneeded input variable x, and is called (Left) *Weakening*, so called because it introduces a type on the left-hand side of the turnstile. Most languages admit weakening, but interweave it throughout the typing rules of their language. We make it explicit here because it is fundamental to one of the effects that form the central example of this paper.

Comp programs can be interpreted either strictly or lazily, and these interpretations can lead to different results. Let us take a look at an example:

$$x \coloneqq 3; y \coloneqq \mathbf{error}; z \coloneqq x$$

In strict semantics, assignments are evaluated from left to right. Thus, the example program assigns 3 to x, and then evaluates the **error** assigned to y. This causes the program to throw an error, stepping immediately to $z \coloneqq \mathbf{error}$ as the output of the program.

In lazy semantics, an assignment is only evaluated when a variable is *needed*, rather than each assignment being evaluated in left-to-right order. Thus, the program looks at the assignment to the final variable, z, and notices that x is needed to compute the final result. It then evaluates $x \coloneqq 3$ and assigns 3 to z. At this point, z no longer needs any other variables to compute its final value, so the whole program steps immediately to $z \coloneqq 3$ without executing any other assignments. Notably, since y is not needed at any point, the assignment of **error** to y is never executed, keeping the error from being thrown. (This form of laziness

$$
\begin{array}{rcl}
x \coloneqq c;\ p^+ & \to_s & p^+[x \mapsto c] \\
x \coloneqq y;\ p^+ & \to_s & p^+[x \mapsto y] \\
x \coloneqq \mathbf{error};\ p;\ y \coloneqq e & \to_s & y \coloneqq \mathbf{error}
\end{array}
$$

Figure 2.3: Strict Comp Reduction Rules

$$
\begin{array}{rcl}
p^+;\ x \coloneqq c & \to_\ell & x \coloneqq c \\
p;\ x \coloneqq e;\ p';\ y \coloneqq x & \to_\ell & p;\ p'[x \mapsto e];\ y \coloneqq e \\
p^+;\ x \coloneqq \mathbf{error} & \to_\ell & x \coloneqq \mathbf{error}
\end{array}
$$

Figure 2.4: Lazy Comp Reduction Rules

is *call-by-need*, which is equivalent to the more-common call-by-name [Ariola et al., 1995; Maraist et al., 1995] in this case.)

We can formalize these two interpretations using the reduction rules in Figures 2.3 and 2.4. In Figure 2.3 we have the strict rules, which go through a program from left to right, substituting variables with values, and jumping to the end when an **error** is encountered. The lazy rules, in Figure 2.4, go through the program from right to left, only substituting variables when needed for the final result.

Since Comp lacks functions, it might seem surprising that Comp can have differing strict and lazy interpretations of programs. One of the central results of this paper is that strictness and laziness can be described as arising from the interaction of effects during program composition. In previous works, programs were composed through function application [Ariola et al., 1995; Levy, 2001; López-Fraguas et al., 2007; Maraist et al., 1995; Plotkin, 1975; Sabry and Wadler, 1997; Wadler, 2003]. In Comp, programs are composed by being set next to each other and joined by a semicolon.

The semicolon-based syntax for composition also makes the connection between

Comp programs and category theory clear. Category theory is used to study the compositional structure of languages. This makes it useful in our goal of describing strictness and laziness as arising from the interactions of effects during composition. Comp types and programs form a category, although some care must be taken because Comp programs have multiple inputs. Technically this requires monoidal category theory [Bénabou, 1963; Mac Lane, 1963], multicategory theory [Lambek, 1969; Leinster, 1998], or (in order to generalize to multiple outputs) polycategory theory [Szabo, 1975]. This generalization is straightforward, but requires much technical detail, so we do not present it here.

As a refresher, a category $\mathbf{D}$ is a collection $|\mathbf{D}|$ of *objects* along with, for every pair of objects a and $b \in |\mathbf{D}|$, a collection $\mathbf{D}(a, b)$ of *morphisms*. For a morphism $f \in \mathbf{D}(a, b)$, we say that a is the domain of f and that b is the codomain of f. Categories must have an identity morphism $\mathtt{id}_a \in \mathbf{D}(a, a)$ for every object $a \in |\mathbf{D}|$. Moreover, it must be possible to compose morphisms, so that if $f \in \mathbf{D}(a, b)$ and $g \in \mathbf{D}(b, c)$, then there is a morphism $f; g \in \mathbf{D}(a, c)$.[1] This composition operation must be associative (so $(f; g); h = f; (g; h)$), and the identity morphism must be an identity for composition (so $\mathtt{id}_a; f = f = f; \mathtt{id}_b$ for any $f \in \mathbf{D}(a, b)$). We write $f : a \to b$ for $f \in \mathbf{D}(a, b)$. In Section 2.4, we use this connection to category theory to give formal semantics to effectful languages.

2.2 Consumer Choice and Producer Choice

We have seen that Comp programs can be interpreted either strictly or lazily, and that this indeed leads to different results. Now, let us investigate why. Consider

[1] Note that we use diagram-order composition $f; g$ instead of function-order composition $g \circ f$.

again the example Comp program

$$x \coloneqq 3; y \coloneqq \textbf{error}; z \coloneqq x$$

Reviewing our earlier reasoning, we can pinpoint why the strict interpretation of this program and the lazy interpretation of this program are different. The expression **error** is assigned to y, which is not used to compute the final result. This means that the lazy reduction rules will not throw an error, since they will never evaluate the assignment to y. Conversely, the strict reduction rules will throw an error, since they evaluate each assignment regardless of whether it is needed.

In general, the programs for which the strict and lazy interpretations lead to different results are exactly those with **error**s that are not used to compute the final result. These programs will have the **error** discarded by the lazy interpretation, so no error is ever thrown. However, the strict interpretation never throws away any assignment, so an error will be thrown.

Looking at the typing proofs for our example program, we can see that there are typing rules that tell us exactly when these two concepts are in play.

$$\dfrac{\dfrac{}{\vdash x \coloneqq 3 \dashv x : \mathbb{N}} \qquad \dfrac{\dfrac{}{\vdash y \coloneqq \textbf{error} \dashv y : \mathbb{N}}\ \textsc{Error} \qquad \dfrac{\dfrac{}{x : \mathbb{N} \vdash z \coloneqq x \dashv z : \mathbb{N}}}{x : \mathbb{N}, y : \mathbb{N} \vdash z \coloneqq x \dashv z : \mathbb{N}}\ \textsc{Weakening}}{x : \mathbb{N} \vdash y \coloneqq \textbf{error}; z \coloneqq x \dashv z : \mathbb{N}}}{\vdash x \coloneqq 3; y \coloneqq \textbf{error}; z \coloneqq x \dashv z : \mathbb{N}}$$

Notice the two rules we have labeled: Error and Weakening. The Error rule tells us that an **error** is assigned to a variable, while the Weakening rule tells us that a variable is discarded.

Note that neither the Error rule nor the Weakening rule is reflected in the types of the program, since the only type in Comp is $\mathbb{N}$. Instead, they tell us

something about the internals of the program. The ERROR rule tells us that some output is never actually provided. The WEAKENING rule tells us that some input is never actually used.

Both rules correspond to effects that are actually fundamental to strictness and laziness. To see this, note that for any program needing at most one of the two rules/effects, the strict and lazy interpretations lead to the same result.

The effect of not providing an output we call *producer choice* (of quantity) because it is the ability for a program to choose how many times it will provide an output. In the case of Comp, programs are limited to choosing to produce an output zero times or one time. The effect of dropping an input we call *consumer choice* (of quantity) because it is the ability for a program to choose how many times it will consume an input. In Comp, programs can choose to use inputs as many times as they want. The bottom rule in Figure 2.2, known as (LEFT) *Contraction*, allows a program to use a variable more than once, another form of consumer choice.

Producer choice describes how output is produced. We refer to effects that describe how output is produced as *producer effects*. One usually uses a categorical construct called a *monad* to give semantics to producer effects [Moggi, 1989; Wadler and Thiemann, 1998]. That is, if $\mathbf{D}$ is a category (such as **Set** or **CPO**) in which pure programs in a language can be interpreted, producer-effectful programs can be intepreted as morphisms in $\mathbf{D}$ with codomain Mb, where M is a monad and b is some type. A monad M on a category $\mathbf{D}$ is a function on the objects of $\mathbf{D}$ along with two operators. The first operator is called the unit of the monad, and is written η. It specifies a morphism $\eta_a : a \to Ma$ for each object $a \in |\mathbf{D}|$. The second operator is called `bind`, and it maps each morphism $f : a \to Mb$ to a

morphism $\texttt{bind}(f) : Ma \to Mb$. These must satisfy the following equations:

- $\texttt{id}_{Ma} = \texttt{bind}(\eta_a)$ for all $a \in |\mathbf{D}|$
- $\eta_a; \texttt{bind}(f) = f$ for all $f : a \to Mb$
- $\texttt{bind}(f); \texttt{bind}(g) = \texttt{bind}(f; \texttt{bind}(g))$ for all $f : a \to Mb$ and $g : b \to Mc$

Consumer choice does not describe how output is produced, but rather it describes how input is consumed. Thus, rather than being a producer effect, it is a *consumer effect*. Consumer effects are usually given semantics through a categorical construct called a comonad [Brookes and Geva, 1992; Petricek et al., 2012, 2014; Uustalu and Vene, 2005, 2008], which is the dual of a monad. That is, if $\mathbf{D}$ is a category in which pure programs in some language can be interpreted, consumer-effectful programs can be interpreted as morphisms in $\mathbf{D}$ with domain Ca, where C is a comonad and a is some type. Formally, a comonad C on a category $\mathbf{D}$ is a function on the objects of $\mathbf{D}$ along with counit and `cobind` operations. The counit, written ϵ, specifies a morphism $\epsilon_a : Ca \to a$ for each object $a \in |\mathbf{D}|$, while `cobind` maps each morphism $f : Ca \to b$ to a morphism $\texttt{cobind}(f) : Ca \to Cb$. These must satisfy the following equations:

- $\texttt{id}_{Ca} = \texttt{cobind}(\epsilon_a)$ for all $a \in |\mathbf{D}|$
- $\texttt{cobind}(f); \epsilon_b = f$ for all $f : Ca \to b$
- $\texttt{cobind}(f); \texttt{cobind}(g) = \texttt{cobind}(\texttt{cobind}(f); g)$ for all $f : Ca \to b$ and $g : Cb \to c$

Producer choice is often given semantics through the `Maybe` monad. Values of type `Maybe` t are either `Nothing`, representing a choice not to provide an output, or

`Some` v, where v is of type t, representing a choice to produce an output. However, it is more difficult to give a simple description of a comonad for consumer choice. Instead of attempting to do so computationally, we first turn to the world of logic. We will see that classical linear logic contains a comonad representing consumer choice and a complementary monad representing producer choice. By studying the comonad and the monad of classical linear logic, we can develop a comonad that represents consumer choice in Comp and a monad that represents producer choice in Comp.

2.3 Capturing Consumer Choice and Producer Choice

The weakening and contraction rules of Comp are inspired by similar rules from sequent calculus for classical logic, also called (Left) Weakening and Contraction [Gentzen, 1935a,b]. Just as Weakening and Contraction provide consumer choice in Comp, classical logic's versions of Left Weakening and Contraction provide consumer choice in classical logic. It also has a *Right* Weakening rule, which inspired the Error rule of Comp since, in Comp, errors produce no output. This rule provides producer choice in classical logic. In fact, classical logic also has a *Right* Contraction rule that allows outputs to be produced more than once, just as the Left Contraction rule allows Comp (and classical logic) to use an input more than once. This expands the capabilities of producer choice in classical logic compared to producer choice in Comp.

On the other hand, (classical) *linear* logic is pure with respect to both consumer choice and producer choice [Girard, 1987]. That is, it has no Contraction or Weakening rules. Instead, linear logic provides a comonad ! (which is

pronounced "bang" or "of course") to capture consumer choice in classical logic, and a monad ? (which is pronounced "query" or "why not") to capture producer choice in classical logic [Girard, 1987].

We framed our discussion of capturing effects around categories, and we can see both classical logic and linear logic as categories. For classical logic, the objects of this category are formulae of classical logic, and the morphisms are (equivalence classes of) classical-logic proofs. For linear logic, the definition of the category is similar but uses formulae and proofs from linear logic instead. As mentioned above, we represent proofs using Gentzen's [1935a; 1935b] sequent calculus.

Recall that a sequent is a pair of multisets of formulae Γ and Δ, written $\Gamma \vdash \Delta$. In classical logic, we interpret this as saying that if all of the assumptions in Γ are true, then at least one of the conclusions in Δ is true. In linear logic, we treat that same sequent as a process that, given all of the resources in Γ, provides all of the resources in Δ. Formulae in linear logic can be treated as denoting resources rather than truth values because linear logic does not have producer choice or consumer choice. In some sense, choice (of quantity) enforces the idea that classical-logic formulae denote truth values. After all, the fact that a formula is true does not become invalid once someone chooses to rely upon that fact. However, providing choice (of quantity) on an arbitrary resource would allow processes to reproduce and exhaust that resource without limit.

A category is more than a collection of objects and morphisms. In order to form a proper category, we need a logical notion of identity, which is provided by Axiom, and of composition, which is provided by Cut. The Axiom rule says that every hypothesis implies itself, or that every resource produces itself. The Cut rule says that a φ provided by one proof can be supplied to another proof.

$$\textsc{Axiom}\ \frac{}{\varphi \vdash \varphi} \qquad \textsc{Cut}\ \frac{\Gamma \vdash \varphi, \Delta \qquad \Gamma', \varphi \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}$$

In order to be a category, these two rules must further satisfy the identity and associativity laws, and to achieve this one must use a more permissive notion of proof equality. We will not go into the details, but consider cutting a proof with Axiom: this produces a new proof, even though it is conceptually the same as the original proof. Both logics have a procedure called cut elimination, which takes a proof that uses the Cut rule and finds a proof of the same sequent that does not use the Cut rule. This process is non-deterministic, but for linear logic it turns out to still be confluent. Using techniques like cut elimination, albeit after fixing a particular cut-elimination (i.e. reduction) strategy for classical logic, one can develop appropriate notions of proof equality for both logics.

With the principles of classical logic and linear logic in hand, we can proceed to show how linear logic captures consumer choice and producer choice in classical logic. We start with classical logic and show how it exhibits both consumer choice and producer choice. Then we continue with linear logic and show how it defines a comonad and a monad that intuitively allow the same kinds of proofs that classical logic admits using consumer choice and producer choice. We finally briefly discuss how to give semantics to effectful proofs from classical logic using linear logic.

Classical logic enables consumer choice and producer choice via the so-called *structural rules*. There are four structural rules in classical logic arising from two binary choices: left vs. right, and weakening vs. contraction. The Left Weakening rule says that a proof does not need to use all of its assumptions. The Left Contraction rule says that a proof can use its assumptions multiple

times. The Right Weakening rule says that we can prove either ψ or φ if we can prove ψ. The Right Contraction rule says that we can prove φ true if we can prove φ or φ true. A formula can be consumed or produced more than once by using contraction, and can be consumed or produced zero times by using weakening.

Classical Logic

Left Weakening	Left Contraction	Right Weakening	Right Contraction
$\dfrac{\Gamma \vdash \Delta}{\Gamma, \varphi \vdash \Delta}$	$\dfrac{\Gamma, \varphi, \varphi \vdash \Delta}{\Gamma, \varphi \vdash \Delta}$	$\dfrac{\Gamma \vdash \Delta}{\Gamma \vdash \varphi, \Delta}$	$\dfrac{\Gamma \vdash \varphi, \varphi, \Delta}{\Gamma \vdash \varphi, \Delta}$

It would not make sense to have a logic of resources where the structural rules held. For instance, the Right Weakening rule would allow a process to create a resource out of nothing, and Left Contraction would allow a process to use twice as many resources as it was given. However, it might seem strange that Left Weakening and Right Contraction are not allowed. To see why they are not allowed, note that linear logic views debt as a form of resource. This allows it to encode "A implies B" as a debt of an A that, when paid, provides a B. This also means the Left Weakening rule would allow the receiver of some debt to choose not to pay it. Similarly, the Right Contraction rule would allow a process to halve the amount of debt it produces.

However, linear logic does allow controlled use of the structural rules through the *exponentials* ! ("bang") and ? ("query"). The "bang" exponential provides the consumer of a resource access to the left structural rules. That is, a formula $!\varphi$ can be duplicated or ignored on the left. The "query" exponential provides the producer of a resource access to the right structural rules. That is, a formula $?\varphi$

can be duplicated or ignored on the right. This is formalized by the following rules:

Classical Linear Logic

Left Weakening	Left Contraction	Right Weakening	Right Contraction
$\dfrac{\Gamma \vdash \Delta}{\Gamma, !\varphi \vdash \Delta}$	$\dfrac{\Gamma, !\varphi, !\varphi \vdash \Delta}{\Gamma, !\varphi \vdash \Delta}$	$\dfrac{\Gamma \vdash \Delta}{\Gamma \vdash ?\varphi, \Delta}$	$\dfrac{\Gamma \vdash ?\varphi, ?\varphi, \Delta}{\Gamma \vdash ?\varphi, \Delta}$

One also eventually wants to actually produce a $!\varphi$ or consume a $?\varphi$. Linear logic includes the rules Right Promotion, which produces a $!\varphi$, and Left Promotion, which consumes a $?\varphi$.[2]

Right Promotion	Left Promotion
$\dfrac{!\Gamma \vdash \varphi, ?\Delta}{!\Gamma \vdash !\varphi, ?\Delta}$	$\dfrac{!\Gamma, \varphi \vdash ?\Delta}{!\Gamma, ?\varphi \vdash ?\Delta}$

Note that the promotion rules restrict their contexts. To see why, imagine that p is a sequent of the form $\Gamma \vdash \varphi, \Delta$. That is, p is a process that uses Γ to produce φ and Δ. Imagine also that q is a sequent of the form $\Gamma', !\varphi \vdash \Delta'$. One might naïvely promote p and compose it with q along $!\varphi$ to get a judgment of the form $\Gamma, \Gamma' \vdash \Delta, \Delta'$. Suppose q chooses to use Left Weakening to indicate that it does not want the $!\varphi$ resource, so the promotion of p must not provide that resource. In linear logic, the only way to do this is simply to not execute p. However, this means that Γ is never consumed and Δ is never produced. Thus, all resources in Γ need to be ! resources and all resources in Δ need to be ? resources so that we can choose not to consume or produce them.

Contrast this with the final rules involving the exponentials, called the dereliction rules.

[2]We use the notation $!\Gamma = \{!\psi \mid \psi \in \Gamma\}$ and $?\Delta = \{?\psi \mid \psi \in \Delta\}$.

Left Dereliction	Right Dereliction
$\dfrac{\Gamma, \varphi \vdash \Delta}{\Gamma, !\varphi \vdash \Delta}$	$\dfrac{\Gamma \vdash \varphi, \Delta}{\Gamma \vdash ?\varphi, \Delta}$

Left Dereliction allows a proof to use a $!\varphi$ by using a single copy of φ. Similarly, Right Dereliction allows a proof of $?\varphi$ to produce a single copy of φ. Here, there are no restricted contexts. Instead of needing to respond to the choices of other proofs, these rules represent making a choice.

Note that the dereliction and promotion rules are exactly the rules that we need to make ! a comonad and ? a monad. We can construct the counit for ! and the unit for ? using dereliction. Similarly, we can construct the `cobind` for ! and the `bind` for ? using promotion.

Counit	Unit	Cobind	Bind
$\dfrac{\dfrac{}{\varphi \vdash \varphi}}{!\varphi \vdash \varphi}$	$\dfrac{\dfrac{}{\varphi \vdash \varphi}}{\varphi \vdash ?\varphi}$	$\dfrac{!\varphi \vdash \psi}{!\varphi \vdash !\psi}$	$\dfrac{\varphi \vdash ?\psi}{?\varphi \vdash ?\psi}$

Using cut elimination, one can prove that these constructions satisfy the comonad and monad laws.

At this point, we have a comonad ! and a monad ? that seem to capture the ideas of consumer choice and producer choice, respectively. However, we only have an informal computational interpretation, whereas we need to have a formal computational interpretation to apply them to Comp. Moreover, we previously discussed giving semantics to effectful programs by giving semantics to pure programs in some category, and then interpreting effectful programs using either a monad or a comonad. So in order to give semantics to classical-logic proofs as

effectful programs, we must be able to give pure proofs—that is, proofs that do not use weakening or contraction—semantics in some category. That category is the category of linear-logic proofs.

It is not difficult to show how to embed pure classical-logic proofs into linear logic. However, we must then be able to use the comonad ! and the monad ? to embed classical-logic proofs that do use weakening or contraction into linear logic. As we discussed in the introduction, there are at least three ways to give semantics to a program with two effects, where one of the effects is given semantics via a comonad and the other is given semantics via a monad. The first uses a distributive law to describe the interactions of effects when programs are composed. The other two are the layerings, which do not use a distributive law.

Girard [1987], who invented linear logic, tried to embed classical-logic proofs into linear logic. However, he was only able to succeed with cut-free proofs. This is quite unsatisfying since, in programming-language terms, this is the equivalent of only being able to translate values. He was unable to embed proofs containing Cut because his method of embedding classical logic into linear logic corresponds to the technique for giving semantics with both kinds of effects that uses distributive laws. However, there is no distributive law between ! and ?, as we will see shortly, and so he had no way to compose, i.e. cut-eliminate, his translations.

Eventually, Girard was able to give a denotational semantics of classical logic using the semantics of classical linear logic using a technique known as *polarization* [Girard, 1991]. However, syntactic embeddings of classical logic into classical linear logic were not provided until Schellinx [1994] was able to give two embeddings of classical logic into classical linear logic *including* proofs containing Cut. His methods correspond to the layerings. He was able to embed proofs that con-

tained Cut because the layerings do not rely on a distributive law for composition. Consequently, one can amusingly view classical logic as effectful linear logic.

2.4 Effectful Languages and Their Semantics

In this section, we develop a linguistic metatheory for languages with two effects, where one effect can be given semantics using a monad and the other effect can be given semantics using a comonad. In order to develop such a metatheory, we focus on languages that can express the monad and comonad that give semantics to their effects. However, the categorical constructions are the same when the language, like Comp, cannot internally express the monad and the comonad.

We start our discussion of effectful languages by looking at languages with just one effect. The semantics we will be interested in here are standard, having been studied at length [Brunel et al., 2014; Filinski, 2010; Marino and Milstein, 2009; Moggi, 1989; Petricek et al., 2012, 2014; Tate, 2013; Wadler and Thiemann, 1998]. However, it is not common to see the linguistic assumptions made explicit, so this should be of some interest even to seasoned experts.

We then move on to discuss languages with two effects. The semantics here are less-commonly discussed, although they have been discovered before [Brookes and van Stone, 1993; Gaboardi et al., 2016; Power and Watanabe, 2002]. Furthermore, we discuss the novel linguistic metatheory of languages with two effects.

Note that in the metatheory we present here, effects ε are not necessarily part of the types τ; instead, they are in a sense an orthogonal classification of programs. In particular, whereas types describe the kind of data that comes in and out of a

$$\frac{}{\mathtt{id}_\tau : \tau \to \tau} \qquad \frac{p : \tau_1 \to \tau_2 \qquad q : \tau_2 \to \tau_3}{p; q : \tau_1 \to \tau_3} \qquad \frac{p : \tau_1 \to \tau_2}{p : \tau_1 \xrightarrow{\varepsilon} \tau_2}$$

$$\frac{p : \tau_1 \xrightarrow{\varepsilon} \tau_2 \qquad q : \tau_2 \xrightarrow{\varepsilon} \tau_3}{p; q : \tau_1 \xrightarrow{\varepsilon} \tau_3}$$

$$\frac{p : \tau_1 \xrightarrow{\varepsilon} \tau_2}{\mathtt{id}_{\tau_1}; p = p} \qquad \frac{p : \tau_1 \xrightarrow{\varepsilon} \tau_2}{p; \mathtt{id}_{\tau_2} = p} \qquad \frac{p : \tau_1 \xrightarrow{\varepsilon} \tau_2 \qquad q : \tau_2 \xrightarrow{\varepsilon} \tau_3 \qquad r : \tau_3 \xrightarrow{\varepsilon} \tau_4}{p; (q; r) = (p; q); r}$$

Figure 2.5: Rules of Effectful Languages

program, effects describe the internal process of the program. Of course, there is sometimes an interplay between types and effects and, as we will demonstrate, the precise form of this interplay often dictates whether the effect is a *producer* effect (i.e. a monadic effect) or a *consumer* effect (i.e. a comonadic effect).

2.4.1 Singly-Effectful Languages

We refer to languages with only one effect as *singly-effectful*. These include languages where effects can be captured by either a monad or a comonad. However, capturing effects in either way requires linguistic assumptions beyond having effectful programs. The core linguistic assumption of singly-effectful programs is that some programs are effectful, while others are pure. To denote this, we write $p : \tau_1 \to \tau_2$ when p is a pure program with input type τ_1 and output type τ_2, and we write $p : \tau_1 \xrightarrow{\varepsilon} \tau_2$ when p is effectful. The pure programs form a sublanguage of the effectful language, meaning we can always consider a pure program $p : \tau_1 \to \tau_2$ as an effectful program $p : \tau_1 \xrightarrow{\varepsilon} \tau_2$. The fact that pure programs form a sublanguage, rather than just a subset of programs, means that pure programs are

closed under composition and that the identity programs are pure. Similarly, the effectful programs are also closed under composition, and the effectful identity is the same as the pure identity. We call any language that admits at least the rules in Figure 2.5 "effectful."

Note that Comp could intuitively be considered singly-effectful. We might consider any Comp programs that never need the ERROR rule pure, while those that do are effectful. Alternatively, we might consider any Comp programs that never need the WEAKENING rule pure, while those that do are effectful. We could even consider a Comp program pure only if it is pure under both definitions, and effectful if it is effectful under either. In order to formalize this intuition, we would have to consider languages where types are actually typing *contexts*, such as those used in Comp. This requires the extra structure of monoidal categories [Bénabou, 1963; Mac Lane, 1963] or multicategories [Lambek, 1969; Leinster, 1998]. This generalization is straightforward, but requires much technical detail, so we do not present it here.

However, considering Comp merely as an effectful language does not give us much of a semantic "handle" on the language. The only semantic outcome of a singly-effectful language is a category of pure programs, and a category of effectful programs, with an embedding of the pure programs into the effectful programs. In order to get monads and comonads, we have to delve deeper.

Producer Effects

A language with a producer effect is one that, in addition to admitting the rules in Figure 2.5, is able to have effectful programs be "thunked," turning them into pure programs. Specifically, if p is an effectful program with type $\tau_1 \xrightarrow{\varepsilon} \tau_2$, then there

$$\frac{p : \tau_1 \xrightarrow{\varepsilon} \tau_2}{\lfloor p \rfloor : \tau_1 \to M\tau_2} \qquad \frac{}{\texttt{exec}_\tau : M\tau \xrightarrow{\varepsilon} \tau} \qquad \frac{p : \tau_1 \xrightarrow{\varepsilon} \tau_2}{\lfloor p \rfloor \,; \texttt{exec}_{\tau_2} = p} \qquad \frac{p : \tau_1 \to M\tau_2}{\lfloor p; \texttt{exec}_{\tau_2} \rfloor = p}$$

Figure 2.6: Rules of Producer Effectful Languages

is some pure program $\lfloor p \rfloor : \tau_1 \to M\tau_2$, where M is some function on types. For instance, if ε is the effect "might throw an error," then M is `Maybe`. If p throws an error, then $\lfloor p \rfloor$ returns `Nothing`. Otherwise if p returns v, then $\lfloor p \rfloor$ returns `Some` v.

Intuitively, if $\lfloor p \rfloor$ returns a value of type $M\tau$, then that value captures all of the effects that would happen if we were to run p. Consequently, producer-effectful languages have an effectful program $\texttt{exec}_\tau : M\tau \xrightarrow{\varepsilon} \tau$ that runs the effects captured in $M\tau$. Thus, if ε is the effect "might throw an error," $\texttt{exec}_\tau$ will throw an error if its input is `Nothing`, and will return v if its input is `Some` v.

We formalize the rules for producer effects in Figure 2.6. Any language that admits these rules in addition to the rules in Figure 2.5 is producer-effectful, and ε is a producer effect. This "thunk-and-exec" view of effects comes from Tate [2013] and is analogous to the "reify-and-reflect" view of Filinski [1999, 2010].

M must be a Monad It is possible to formally show that M must be a monad in any producer-effectful language. We need to build η and `bind`, and to show that the equations of a monad hold. To build η, we use $\lfloor \texttt{id}_\tau \rfloor$. Recall that $\texttt{id}_\tau : \tau \to \tau$ and that pure programs can be turned into effectful programs, so $\texttt{id}_\tau : \tau \xrightarrow{\varepsilon} \tau$. Thus, $\lfloor \texttt{id}_\tau \rfloor$ has signature $\tau \to M\tau$, as desired. For *bind*, we have to do something more complicated: $\texttt{bind}(f) = \lfloor \texttt{exec}_{\tau_1}; f; \texttt{exec}_{\tau_2} \rfloor$, where $f : \tau_1 \to M\tau_2$. This executes its input effects, runs f, and then executes the output of f, essentially combining the effects before capturing them all in one large thunk. The equations

for monads follow from the equations for thunks and `exec`.

Structure for Producer-Effectful Programs Every producer-effectful program $f : \tau_1 \xrightarrow{\varepsilon} \tau_2$ corresponds to a morphism $\lfloor f \rfloor : \tau_1 \to M\tau_2$ in the category of pure programs. It is also a small exercise to show that $\lfloor p; q \rfloor = \lfloor p \rfloor ; \texttt{bind}(\lfloor q \rfloor)$ for any producer-effectful programs p and q. This shows that the category of producer-effectful programs is the Kleisli category for M, denoted K_M. We formally define K_M for a monad M on a category $\mathbf{D}$ as follows:

- The objects of K_M are the same as the objects of $\mathbf{D}$.
- The morphisms from a to b in K_M are the morphisms from a to Mb in $\mathbf{D}$.
- The identity morphisms $\texttt{id}_a : a \to a$ in K_M are the unit of the monad $\eta_a : a \to Ma$ in $\mathbf{D}$.
- For any $f : a \to b$ and $g : b \to c$ in K_M, the composition $f; g$ in K_M is $f; \texttt{bind}\,(g)$ in $\mathbf{D}$.

Theorem 1. *Let $\mathcal{L}$ be a producer-effectful language, and let $\mathbf{D}$ be a category that gives semantics to the pure programs in $\mathcal{L}$. Let M be a monad on $\mathbf{D}$ corresponding to the function on types M with appropriate unit and bind. Then K_M gives semantics to the producer-effectful programs in $\mathcal{L}$.*

Furthermore, the pure programs in $\mathcal{L}$ form a category $\mathbf{D}$, and the function on types M forms a monad on that category. The Kleisli category K_M corresponding to this choice of $\mathbf{D}$ and M is isomorphic to the category of producer-effectful programs in $\mathcal{L}$.

In fact, the requirements of the first half of Theorem 1 are stronger than necessary. The monad M does not need to be expressible within the language $\mathcal{L}$ itself, it just needs to have appropriate corresponding structure on $\mathbf{D}$. Such a situation is more in line with how Moggi [1989] and Wadler and Thiemann [1998] use Kleisli categories to give semantics to effectful languages. The second half of the theorem, which states that producer effects are necessarily monadic, comes from Tate [2013] and does require that the function on types M can be expressed within $\mathcal{L}$.

This distinction is important, particularly for giving semantics to Comp. The type system of Comp is too weak to express the necessary function on types, since it has only one type $\mathbb{N}$. Thus, later in the paper, we will develop another language, Proc, that is pure but has a more-expressive type system that is capable of expressing the functions on types that capture the effects in Comp.

Consumer Effects

While producer effects have thunking that changes only the output types, consumer effects have thunking that changes only the input types. That is, given a consumer-effectful program $p : \tau_1 \xrightarrow{\varepsilon} \tau_2$ there is some pure program $\lceil p \rceil : C\tau_1 \to \tau_2$ for some function on types C. For instance, if p can use some extra information of type S, then $\lceil p \rceil$ has signature $\tau_1 \times S \to \tau_2$. This program simply looks at the second component of its input whenever p uses that information. We can think of consumer effects as "needing something extra," while producer effects "make something extra." When ε is "may use extra information,", i.e. may read some immutable state, this is very direct: the environment provides an extra input of type S that p can use.

When we introduced producer effects, we had an intuition that M captured all

$$\frac{p : \tau_1 \xrightarrow{\varepsilon} \tau_2}{\lceil p \rceil : C\tau_1 \to \tau_2} \qquad \frac{}{\texttt{coexec}_\tau : \tau \xrightarrow{\varepsilon} C\tau} \qquad \frac{p : \tau_1 \xrightarrow{\varepsilon} \tau_2}{\texttt{coexec}_{\tau_1}; \lceil p \rceil = p} \qquad \frac{p : C\tau_1 \to \tau_2}{\lceil \texttt{coexec}_{\tau_1}; p \rceil = p}$$

Figure 2.7: Rules of Consumer Effectful Languages

of the effects in an effectful program, and that we could therefore execute M to get those effects back. A similar intuition holds here: C captures all of the information that p needs to run. *Co-execution*, which we denote via a program $\texttt{coexec}_\tau : \tau \xrightarrow{\varepsilon} C\tau$, performs effectful operations in order to capture the information necessary for C. In the case of using extra information, co-execution takes a value v, reads the state to get s, and then returns $(v, s) : \tau \times S$.

We formalize what it means to be a consumer effect in Figure 2.7. Any language that admits these rules in addition to the rules in Figure 2.5 is consumer-effectful, and ε is a consumer effect.

C must be a Comonad To formally show that C must be a comonad in any consumer-effectful language, we construct ϵ_τ as $\lceil id_\tau \rceil$, and $\texttt{cobind}(f)$ as $\lceil \texttt{coexec}_{\tau_1}; f; \texttt{coexec}_{\tau_2} \rceil : C\tau_1 \to C\tau_2$ for any pure program $f : C\tau_1 \to \tau_2$. In the case of using extra information, $\epsilon_\tau : \tau \times S \to \tau$ takes a value of the form (v, s) and returns v. In the same setting, $\texttt{cobind}(p)$ takes a value of the form (v, s) and runs $p(v, s)$ to get v', and then returns (v', s).

Structure for Consumer-Effectful Programs We can also show that the category of consumer-effectful programs is structured as the Kleisli category for C, written K_C. We formally define K_C for a comonad C on a category $\mathbf{D}$ as follows:

- The objects of K_C are the same as the objects of $\mathbf{D}$.
- The morphisms from a to b in K_C are the morphisms from Ca to b in $\mathbf{D}$.
- The identity morphisms $\texttt{id} : a \to a$ in K_C are the counit of the comonad $\epsilon_a : Ca \to a$ in $\mathbf{D}$.
- For any $f : a \to b$ and $g : b \to c$ in K_C, the composition $f; g$ in K_C is $\texttt{cobind}(f); g$ in $\mathbf{D}$.

Theorem 2. *Let $\mathcal{L}$ be a consumer-effectful language, and let $\mathbf{D}$ be a category that gives semantics to the pure programs in $\mathcal{L}$. Let C be a comonad on $\mathbf{D}$ corresponding to the function on types C with appropriate counit and cobind. Then K_C gives semantics to the consumer-effectful programs in $\mathcal{L}$.*

Furthermore, the pure programs in $\mathcal{L}$ form a category $\mathbf{D}$, and the function on types C forms a comonad on that category. The Kleisli category K_C corresponding to this choice of $\mathbf{D}$ and C is isomorphic to the category of consumer-effectful programs in $\mathcal{L}$.

Again, the requirements in the first half of Theorem 2 are stronger than necessary. The comonad C does not need to be expressible within the language $\mathcal{L}$ itself; it just needs to have appropriate corresponding structure on $\mathbf{D}$. This situation corresponds more closely to how Uustalu and Vene [2005, 2008] and Petricek, Orchard, and Mycroft [2012, 2014] give meaning to effectful programs. The second half of Theorem 2 does require C to be expressible in $\mathcal{L}$, and follows from dualizing an argument of Tate [2013].

$$\frac{}{\mathtt{id}_\tau : \tau \to \tau} \qquad \frac{p : \tau_1 \xrightarrow{\vec{\varepsilon}} \tau_2 \qquad q : \tau_2 \xrightarrow{\vec{\varepsilon}} \tau_3}{p; q : \tau_1 \xrightarrow{\vec{\varepsilon}} \tau_3} \qquad \frac{p : \tau_1 \xrightarrow{\vec{\varepsilon}} \tau_2}{\mathtt{id}_{\tau_1}; p = p} \qquad \frac{p : \tau_1 \xrightarrow{\vec{\varepsilon}} \tau_2}{p; \mathtt{id}_{\tau_2} = p}$$

$$\frac{p : \tau_1 \xrightarrow{\vec{\varepsilon}} \tau_2 \qquad \varepsilon_p \in \vec{\varepsilon}}{\lfloor p \rfloor : \tau_1 \xrightarrow{\vec{\varepsilon} \setminus \{\varepsilon_p\}} M\tau_2} \qquad \frac{}{\mathtt{exec}_\tau : M\tau \xrightarrow{\varepsilon_p} \tau} \qquad \frac{p : \tau_1 \xrightarrow{\vec{\varepsilon}} \tau_2}{\lfloor p \rfloor ; \mathtt{exec}_{\tau_2} = p}$$

$$\frac{p : \tau_1 \xrightarrow{\vec{\varepsilon}} M\tau_2 \qquad \varepsilon_p \notin \vec{\varepsilon}}{\lfloor p; \mathtt{exec}_{\tau_2} \rfloor = p}$$

$$\frac{p : \tau_1 \xrightarrow{\vec{\varepsilon}} \tau_2 \qquad \varepsilon_c \in \vec{\varepsilon}}{\lceil p \rceil : C\tau_1 \xrightarrow{\vec{\varepsilon} \setminus \{\varepsilon_c\}} \tau_2} \qquad \frac{}{\mathtt{coexec}_\tau : \tau \xrightarrow{\varepsilon_c} C\tau} \qquad \frac{p : \tau_1 \xrightarrow{\vec{\varepsilon}} \tau_2}{\mathtt{coexec}_{\tau_1}; \lceil p \rceil = p}$$

$$\frac{p : C\tau_1 \xrightarrow{\vec{\varepsilon}} \tau_2 \qquad \varepsilon_c \notin \vec{\varepsilon}}{\lceil \mathtt{coexec}_{\tau_1}; p \rceil = p}$$

$$\frac{p : \tau_1 \to \tau_2}{p : \tau_1 \xrightarrow{\vec{\varepsilon}} \tau_2}$$

Figure 2.8: Formalization of Doubly-Effectful Languages

2.4.2 Doubly-Effectful Languages

While we have discussed producer effects and consumer effects in isolation, Comp has one of each kind of effect. To discuss this, we move from singly-effectful languages to *doubly-effectful* languages. A doubly-effectful language has both a producer effect ε_p and a consumer effect ε_c.

We give the linguistic assumptions of doubly-effectful languages in Figure 2.8. In our formalization, every function arrow is labeled with a set of effects, $\vec{\varepsilon}$. For concision, we omit the usual braces delimiting sets. Depending on the effect set of a program, we refer to the program as "pure," "producer-effectful," "consumer-effectful," "singly-effectful," or "doubly-effectful."

Doubly-effectful languages have all of the features of both producer-effectful languages and consumer-effectful languages. Furthermore, thunking has been extended to handle programs with multiple effects. If p has type $\tau_1 \xrightarrow{\varepsilon_c,\varepsilon_p} \tau_2$, then $\lfloor p \rfloor$ has type $\tau_1 \xrightarrow{\varepsilon_c} M\tau_2$ and $\lceil p \rceil$ has type $C\tau_1 \xrightarrow{\varepsilon_p} \tau_2$.

The subsumption rule in Figure 2.8 applies only to pure programs. Notably, subsumption rules for singly-effectful programs are missing. This is because the three methods of giving semantics to doubly-effectful programs differ by which such subsumption rules are admissible.

It is worth noting that the description of the linguistic assumptions of doubly-effectful languages here is novel, as is the discovery that different subsumption rules correspond to different semantics.

Distributive Laws for Doubly-Effectful Languages

A common assumption is that all singly-effectful programs can be considered doubly-effectful. We formalize this assumption with the two subsumption rules in Figure 2.9. These rules can be used to build a *distributive law*. Let $\mathbf{D}$ be a category, with M a monad on $\mathbf{D}$ and C a comonad on $\mathbf{D}$. A distributive law of M over C specifies a morphism $\sigma_a : CMa \to MCa$ for each object $a \in |\mathbf{D}|$. These must satisfy the following equations:

- $\texttt{cobind}(f; \eta_b); \sigma_b = \texttt{cobind}(f); \eta_{Cb}$ for all $f : Ca \to b$
- $\sigma_a; \texttt{bind}(\epsilon_a; f) = \epsilon_{Ma}; \texttt{bind}(f)$ for all $f : a \to Mb$
- $\texttt{cobind}(\sigma_a; \texttt{bind}(f)); \sigma_b = \sigma_a; \texttt{bind}(\texttt{cobind}(f); \sigma_b)$ for all $f : Ca \to Mb$

In fact, there are two candidates for building σ_τ: we could use either of the

programs $\lfloor\lceil \mathtt{exec}_\tau; \mathtt{coexec}_\tau \rceil\rfloor$ or $\lceil\lfloor \mathtt{exec}_\tau; \mathtt{coexec}_\tau \rfloor\rceil$. These come from the intuition that a distributive law is a doubly-effectful program that converts M and C into their respective effects, made pure. Both of these candidates turn out to be the same, thanks to the following:

Lemma 1. *Let p be any program in a distributive doubly-effectful language. Then $\lfloor\lceil p \rceil\rfloor = \lceil\lfloor p \rfloor\rceil$.*

Proof. Both $\lfloor\lceil p \rceil\rfloor$ and $\lceil\lfloor p \rfloor\rceil$ contain the same information. In particular, the following holds (where the well-typedness of the terms requires the subsumption rules in Figure 2.9):

$$\mathtt{coexec}_{\tau_1}; \lfloor\lceil p \rceil\rfloor; \mathtt{exec}_{\tau_2} = \mathtt{coexec}_{\tau_1}; \lceil\lfloor p \rfloor\rceil; \mathtt{exec}_{\tau_2} = p$$

This equality implies $\lfloor\lceil p \rceil\rfloor$ equals $\lceil\lfloor p \rfloor\rceil$ by applying the following implications:

$$\forall q, q' : \tau \xrightarrow{\vec{\varepsilon}} M\tau'. \quad \varepsilon_p \notin \vec{\varepsilon} \quad \Longrightarrow \quad q; \mathtt{exec}_{\tau'} = q'; \mathtt{exec}_{\tau'} \quad \Longrightarrow \quad q = q'$$

$$\forall q, q' : C\tau \xrightarrow{\vec{\varepsilon}} \tau'. \quad \varepsilon_c \notin \vec{\varepsilon} \quad \Longrightarrow \quad \mathtt{coexec}_\tau; q = \mathtt{coexec}_\tau; q' \quad \Longrightarrow \quad q = q'$$

which are provable from the rules in Figure 2.8:

$$q; \mathtt{exec}_{\tau'} = q'; \mathtt{exec}_{\tau'} \quad \Longrightarrow \quad q = \lfloor q; \mathtt{exec}_{\tau'} \rfloor = \lfloor q'; \mathtt{exec}_{\tau'} \rfloor = q'$$

$$\mathtt{coexec}_\tau; q = \mathtt{coexec}_\tau; q' \quad \Longrightarrow \quad q = \lceil \mathtt{coexec}_\tau; q \rceil = \lceil \mathtt{coexec}_\tau; q' \rceil = q' \qquad \square$$

Lemma 1 lets us write $\lfloor p \rceil = \lfloor\lceil p \rceil\rfloor = \lceil\lfloor p \rfloor\rceil$ without ambiguity. Moreover, it means that we have only one candidate for σ_τ, which is $\lfloor \mathtt{exec}_\tau; \mathtt{coexec}_\tau \rceil$. The laws of thunking, executing, and co-executing make this a distributive law.

Semantics Based on Distributive Laws

In order to build a distributive law this way, the doubly-effectful language must admit the rules in Figure 2.9. To see why, consider the composition $\mathtt{exec}_\tau; \mathtt{coexec}_\tau$.

$$\frac{p : \tau_1 \xrightarrow{\varepsilon_p} \tau_2}{p : \tau_1 \xrightarrow{\varepsilon_c, \varepsilon_p} \tau_2} \qquad \frac{p : \tau_1 \xrightarrow{\varepsilon_c} \tau_2}{p : \tau_1 \xrightarrow{\varepsilon_c, \varepsilon_p} \tau_2}$$

Figure 2.9: Subsumption for Distributive Doubly-Effectful Languages

By the rules of Figure 2.8, two programs can be composed only when their effect sets are the same. In order to compose $\texttt{exec}_\tau$ with $\texttt{coexec}_\tau$, we must lift $\texttt{exec}_\tau$ from a producer-effectful program to a doubly-effectful program, and similarly with $\texttt{coexec}_\tau$. The subsumption laws of Figure 2.9 are required to do this.

Because our description of doubly-effectful languages is novel, so is this argument that admitting the "obvious" subsumption rules in Figure 2.9 allows us to build a distributive law. This gives a description of language features that allow the standard distributive-law-based semantics to work.

Just as the monad and comonad laws enable us to describe and compose producer-effectful programs and consumer-effectful programs solely using pure programs, a distributive law enables us to describe and compose doubly-effectful programs solely using pure programs. Given a category $\mathbf{D}$, with a monad M, a comonad C, and a distributive law σ of M over C, we develop a category $K^\sigma_{C,M}$ such that $K^\sigma_{C,M}(a, b) = \mathbf{D}(Ca, Mb)$. The distributive law σ is necessary to build the composition operator for $K^\sigma_{C,M}$. To see why, consider $f : a \to b$ and $g : b \to c$ in $K^\sigma_{C,M}$. Let us write $[f] : Ca \to Mb$ and $[g] : Cb \to Mc$ for f and g considered as morphisms of $\mathbf{D}$. Then, what should $f; g$ be? We can try to do what we did for the Kleisli and Kleisli categories, and get $\texttt{cobind}([f]) : Ca \to CMb$ and $\texttt{bind}([g]) : MCb \to Mc$, but there is still a type mismatch. However, if we have a distributive law σ, we can use it to fix this type mismatch, so that $f; g$ corresponds to $\texttt{cobind}([f]); \sigma_b; \texttt{bind}([g])$. This leads to the following definition of $K^\sigma_{C,M}$:

- The objects of $K^{\sigma}_{C,M}$ are the same as the objects of $\mathbf{D}$.
- The morphisms from a to b in $K^{\sigma}_{C,M}$ are the morphisms from Ca to Mb in $\mathbf{D}$.
- The identity morphisms $\mathtt{id}_a : a \to a$ in $K^{\sigma}_{C,M}$ are defined as $\epsilon_a; \eta_a : Ca \to Ma$ in $\mathbf{D}$.
- For $f : a \to b$ and $g : b \to c$ in $K^{\sigma}_{C,M}$, their composition is $\mathtt{cobind}(f); \sigma_b; \mathtt{bind}(g)$ in $\mathbf{D}$.

Theorem 3. *Let $\mathcal{L}$ be a doubly-effectful language admitting the subsumption rules in Figure 2.9, and let $\mathbf{D}$ be a category that gives semantics to the pure programs in $\mathcal{L}$. Let M be a monad on $\mathbf{D}$ corresponding to the function on types M with appropriate unit and bind, and let C be a comonad on $\mathbf{D}$ corresponding to the function on types C with appropriate counit and cobind. Finally, let σ be a distributive law of M over C. Then $K^{\sigma}_{C,M}$ gives semantics to the doubly-effectful programs in $\mathcal{L}$.*

Furthermore, the pure programs in $\mathcal{L}$ form a category $\mathbf{D}$, the function on types M forms a monad on that category, the function on types C forms a comonad on that category, and the collection of programs $[\mathtt{exec}_\tau; \mathtt{coexec}_\tau]$ forms a distributive law σ of M over C. The category $K^{\sigma}_{C,M}$ corresponding to this choice of $\mathbf{D}$, M, C, and σ is isomorphic to the category of doubly-effectful programs in $\mathcal{L}$.

Yet again, the requirements of the first half of Theorem 3 are stronger than necessary. Weakening them gives a situation similar to that studied by Power and Watanabe [2002], Brookes and van Stone [1993], and Gaboardi, Katsumata, Orchard, Breuvart, and Uustalu [2016]. The second half of Theorem 3 is novel, and it requires $\mathcal{L}$ to admit the subsumption laws in Figure 2.9.

Linear Logic Lacks a Distributive Law

Girard's attempt to embed classical-logic proofs into linear logic embeds classical-logic sequents of the form $\Gamma \vdash \Delta$ as classical-linear-logic sequents of the form $!\Gamma \vdash ?\Delta$. This looks a lot like the development of $K^{\sigma}_{C,M}$. However, Girard was only able to translate *cut-free* proofs of classical logic into linear logic using this method.

Recall that cut is how sequential composition is implemented in logic. Because $K^{\sigma}_{C,M}$ relies on σ to build composition, we might expect that there is a problem with building a distributive law in linear logic. Indeed, there is *no* distributive law of ? over !.

Such a law would be a proof of the sequent $!?\varphi \vdash ?!\varphi$ for any φ. One might expect this to be provable, given that one has free choice over how many inputs they request and how many outputs they provide. In particular, one could opt to request no inputs and provide no outputs. However, this would fail to satisfy the requirements of a distributive law. Unfortunately, the requirements of a distributive law dynamically constrain just how many inputs and outputs need to be provided, and one is inevitably forced to pick how many inputs they request before they know how many outputs they need to provide or vice versa. More formally, the fact that a distributive law cannot exist follows from Schellinx's work [Schellinx, 1994] and the upcoming Theorem 6.

Brookes and van Stone [1993] argue that it is appropriate to use distributive laws in the semantics of effects because, in their exploration of effects, every application had a distributive law.[3] Since then, distributive laws have been the focus of

[3]Brookes and van Stone do find monad-comonad pairs that do not have distributive laws. However, these pairs do not correspond to known computational effects; rather, they are used in

research in giving semantics to doubly-effectful languages [Gaboardi et al., 2016; Power and Watanabe, 2002]. However, the fact that strictness and laziness arise from a pair of effects where the relevant monad and comonad do *not* have a distributive law suggests that exploring other semantics is sometimes necessary.

The Monad-Prioritizing Layering

Consider a strict interpretation of a language where programs have consumer choice, and where producer choice is implemented by throwing errors. In this language, we cannot embed arbitrary producer-effectful programs into the doubly-effectful language. To see why, recall that $\texttt{exec}_\tau$ for throwing errors examines a value v of type `Maybe` τ and, if it is `Some` v, it returns v. Otherwise, v is `Nothing`, and $\texttt{exec}_\tau$ throws an exception. Since it may throw an exception, $\texttt{exec}_\tau$ is producer-effectful.

Suppose we have a program p that may or may not throw an exception, and a program q that may or may not use its input. If we assume subsumption as in Figure 2.9, then we can treat both p and q as doubly-effectful programs and compose them into $p; q$. This program must then be semantically equivalent to $\lfloor p \rfloor ; \texttt{exec}; q$, and therefore to $\lfloor p \rfloor ; \texttt{coexec}; \lceil \texttt{exec}; q \rceil$. Now consider the program $\lceil \texttt{exec}; q \rceil$. It takes as input a $CM\tau$. Furthermore, if p is actually **error** so that the M in $CM\tau$ captures an error, then $p; q$ throws an error by strictness, and so $\lceil \texttt{exec}; q \rceil$ must throw an error due to the established semantic equivalences, and it must only throw an error if p does. So no matter what q does, $\lceil \texttt{exec}; q \rceil$ must extract the $M\tau$ from the $CM\tau$, i.e. apply left dereliction. To see why this is a problem, suppose that p does not throw an exception, so that the $M\tau$ is actually

the study of domain theory.

$$\frac{p : \tau_1 \xrightarrow{\varepsilon_p} C\tau_2}{p : \tau_1 \xrightarrow{\varepsilon_c,\varepsilon_p} C\tau_2} \qquad \frac{p : \tau_1 \xrightarrow{\varepsilon_c} \tau_2}{p : \tau_1 \xrightarrow{\varepsilon_c,\varepsilon_p} \tau_2}$$

Figure 2.10: Subsumption for Strict Doubly-Effectful Languages

a τ, and furthermore suppose that τ is actually some debt, and that q, rather than addressing this debt, has the consumer effect because it explicitly ignores the debt. Then because $\lceil \texttt{exec}; q \rceil$ had to examine the input to determine whether or not to throw an exception, it also forced us to have this debt that is unsoundly left unpaid. Thus it is unsound to allow p and q to be composed together in this strict language, which means we cannot assume subsumption as in Figure 2.9.

Instead, strict languages can only use the weaker subsumption rules in Figure 2.10. The restriction is that producer-effectful programs can only be given the additional consumer effect if they are returning a C value. This C value ensures that subsequent consumer-effectful programs still have a way to discard their input even if the producer-effectful program needs to examine its own. Note that there is no restriction on when consumer-effectful programs can be given the additional producer effect. This might be surprising, but we can use layering to prove that this is sound.

When we gave semantics to doubly-effectful programs via distributive laws, we thunked doubly-effectful programs to pure programs by thunking the effects in either order. That is, $p : \tau_1 \xrightarrow{\varepsilon_c,\varepsilon_p} \tau_2$ was thunked as $[p] : C\tau_1 \to M\tau_2$. However, for strict languages, we thunk in a special way: if $p : \tau_1 \xrightarrow{\varepsilon_c,\varepsilon_p} \tau_2$ is a doubly-effectful program, $[\![p]\!] : C\tau_1 \to MC\tau_2$ is the thunked program where the inner C in the output type enables subsequent programs to drop their inputs even after examining the effects captured in the M. We define $[\![p]\!]$ as $[p; \texttt{coexec}_{\tau_2}]$[4].

[4]This notation is unambiguous here because the proof of Lemma 1 can be adapted to programs

We would like to give structure to the category of doubly-effectful languages in this setting. We can do so with a similar trick as before: given a category $\mathbf{D}$ with a monad M and a comonad C, we develop a category $K^M_{C,M}$ such that $K^M_{C,M}(a, b) = \mathbf{D}(Ca, MCb)$. Moreover, composition is simple: we just use `bind`, just as in the Kleisli category for a monad. This leads to the following definition of $K^M_{C,M}$:

- The objects of $K^M_{C,M}$ are the same as the objects of $\mathbf{D}$.
- The morphisms from a to b in $K^M_{C,M}$ are the morphisms from Ca to MCb in $\mathbf{D}$.
- The identity morphisms $\mathtt{id}_a : a \to a$ in $K^M_{C,M}$ are defined as η_{Ca} in $\mathbf{D}$.
- For any $f : a \to b$ and $g : b \to c$ in $K^M_{C,M}$, their composition is $f; \mathtt{bind}(g)$ in $\mathbf{D}$.

Note that `bind` and not `cobind` is used in the definition of composition, so the producer effect will always be in control of the composition. For instance, if M is `Maybe`, and p throws an error, then the entirety of $p; q$ will throw an error because $[\![p; q]\!] = [\![p]\!]; \mathtt{bind}([\![q]\!])$, making the latter half of the program propagate the error. For that reason, we refer to this as the monad-prioritizing or strict semantics for the effects.

Theorem 4. *Let $\mathcal{L}$ be a doubly-effectful language admitting the subsumption rules in Figure 2.10, and let $\mathbf{D}$ be a category that gives semantics to the pure programs in $\mathcal{L}$. Let M be a monad on $\mathbf{D}$ corresponding to the function on types M with appropriate unit and bind, and let C be a comonad on $\mathbf{D}$ corresponding to the function on types C with appropriate counit and cobind. Then $K^M_{C,M}$ gives semantics to the doubly-effectful programs in $\mathcal{L}$.*

in strict doubly-effectful languages with output types of the form $C\tau$.

Furthermore, the pure programs in $\mathcal{L}$ form a category $\mathbf{D}$, the function on types M forms a monad on that category, and the function on types C forms a comonad on that category. The category of doubly-effectful programs in $\mathcal{L}$ is a subcategory of the category $K^M_{C,M}$ corresponding to this choice of $\mathbf{D}$, M, and C.

Weakening Theorem 4 gives a situation similar to that studied by Brookes and van Stone [1993]. However, they did not apply this to effectful languages, and so did not discover the distributive laws in Figure 2.10. Power and Watanabe [2002] also studied $K^M_{C,M}$ from a purely category-theoretic perspective, but they did not connect to languages at all.

We need to be careful here: the category of doubly-effectful programs is *not* $K^M_{C,M}$. Instead, it is a subcategory of $K^M_{C,M}$. To see why, consider the case where ε_c is reading state, and ε_p is throwing errors. Then a program like $\lambda(x, s).\texttt{Some}\ (x, s+1)$ might exist, and is of type $\tau \times S \to \texttt{Maybe}\ (\tau \times S)$ (where $+1$ is well-defined for S). However, this program does not denote any doubly-effectful program in the language, since it changes the state rather than just reading the state.

The Comonad-Prioritizing Layering

By similar reasoning as in the previous subsection, we cannot treat arbitrary consumer-effectful programs as doubly-effectful when using lazy semantics. However, we can do so when said consumer-effectful programs consume an M. We consequently weaken the subsumption rules of Figure 2.9 to those of Figure 2.11 (as opposed to those of Figure 2.10) for lazy doubly-effectful languages.

We can now thunk doubly-effectful programs of the form $p : \tau_1 \xrightarrow{\varepsilon_c, \varepsilon_p} \tau_2$

$$\frac{p : \tau_1 \xrightarrow{\varepsilon_p} \tau_2}{p : \tau_1 \xrightarrow{\varepsilon_c, \varepsilon_p} \tau_2} \qquad \frac{p : M\tau_1 \xrightarrow{\varepsilon_c} \tau_2}{p : M\tau_1 \xrightarrow{\varepsilon_c, \varepsilon_p} \tau_2}$$

Figure 2.11: Subsumption for Lazy Doubly-Effectful Languages

as $[\![p]\!] : CM\tau_1 \to M\tau_2$ using the construction $[\mathtt{exec}_{\tau_1}; p]$[5], where the M in the input type enables previous programs to not provide their outputs before providing the effects captured in the C. This leads to the following definition of the category $K^C_{C,M}$ for a category $\mathbf{D}$, a monad M on $\mathbf{D}$, and a comonad C on $\mathbf{D}$:

- The objects of $K^C_{C,M}$ are the same as the objects of $\mathbf{D}$.
- The morphisms from a to b in $K^C_{C,M}$ are the morphisms from CMa to Mb in $\mathbf{D}$.
- The identity morphisms $\mathtt{id}_a : a \to a$ in $K^C_{C,M}$ are defined as ϵ_{Ma} in $\mathbf{D}$.
- For any $f : a \to b$ and $g : b \to c$ in $K^C_{C,M}$, their composition is $\mathtt{cobind}(f); g$ in $\mathbf{D}$.

Note that `cobind` and not `bind` is used in the definition of composition, so the consumer effect will always be in control of the composition. For instance, if C is ! and q drops its input, then the entirety of $p; q$ will drop its input because $[\![p; q]\!] = \mathtt{cobind}([\![p]\!]); [\![q]\!]$, making the former half of the program also drop its input. For that reason, we refer to this as the comonad-prioritizing or lazy semantics for the effects.

Theorem 5. *Let $\mathcal{L}$ be a doubly-effectful language admitting the subsumption rules in Figure 2.11, and let $\mathbf{D}$ be a category that gives semantics to the pure programs in*

[5]This notation is unambiguous here because the proof of Lemma 1 can be adapted to programs in lazy doubly-effectful languages with input types of the form $M\tau$.

$\mathcal{L}$. Let M be a monad on $\mathbf{D}$ corresponding to the function on types M with appropriate unit and bind, and let C be a comonad on $\mathbf{D}$ corresponding to the function on types C with appropriate counit and cobind. Then $K^C_{C,M}$ gives semantics to the doubly-effectful programs in $\mathcal{L}$.

Furthermore, the pure programs in $\mathcal{L}$ form a category $\mathbf{D}$, the function on types M forms a monad on that category, and the function on types C forms a comonad on that category. The category of doubly-effectful programs in $\mathcal{L}$ is a subcategory of the category $K^C_{C,M}$ corresponding to this choice of $\mathbf{D}$, M, and C.

Again, weakening this theorem gives a situation similar to that studied by Brookes and van Stone [1993], although they did not study effectful languages. Power and Watanabe [2002] studied $K^C_{C,M}$ purely categorically, but they did not connect to languages at all.

We need to be careful here: the category of doubly-effectful programs is *not* $K^C_{C,M}$. Instead, it is a subcategory of $K^C_{C,M}$. To see why, consider the case where ε_c is reading state, and ε_p is throwing errors. Then a program of type $(\texttt{Maybe}\ \mathbb{N}) \times S \to \texttt{Maybe}\ \mathbb{N}$ could (regardless of the state) return `Nothing` if the input is `Some` v and otherwise return `Some` 5 if the input is `Nothing`. However, this program does not denote any doubly-effectful program in the language, since it turns errors thrown earlier in the program into successes and successes into errors.

The Layerings in Linear Logic

Schellinx [1994] eventually found two translations of classical logic into linear logic that can handle cuts. These correspond to the layerings. The first translates a classical-logic sequent $\Gamma \vdash \Delta$ into the linear-logic sequent $!\Gamma \vdash ?!\Delta$, and the

second translates the same classical sequent into the linear sequent $!?\Gamma \vdash ?\Delta$. In particular, the first translation uses ?-promotion to translate cuts, and the second translation uses !-promotion. This is directly analogous to the use of `bind` versus `cobind` in the two layerings, so the first translation is conceptually the "strict" translation, while the second is the "lazy" translation.

Note, however, that this requires generalizing the assumptions we have made so far. In particular, linear logic is a multiple-input, multiple-output setting whereas the category theory we have presented is a single-input, single-output setting. The categorical properties of ! and ? that allow them to handle this change in setting can be found in Section 2.5.

The Relationship between Distributive Laws and Layering

Using layering, we can prioritize either the producer effect or the consumer effect. However, the definition of $K^{\sigma}_{C,M}$ prioritizes neither, since it uses a distributive law. We would like to compare these semantics, but they have different types. Luckily, it is relatively easy to use η and ϵ to transform $\llbracket p \rrbracket$ and $\lceil\!\lceil p \rceil\!\rceil$ to have the same type as $[p]$. Theorem 6 states that the three semantics are equivalent *provided a distributive law exists.*

Theorem 6. *For any effectful program $p : \tau_1 \xrightarrow{\varepsilon_c, \varepsilon_p} \tau_2$ in a distributive doubly-effectful language, the following are all equal:*

$$\llbracket p \rrbracket ; \lfloor \mathtt{exec}_{C\tau} ; \lceil \mathtt{id}_{\tau} \rceil \rfloor \quad = \quad [p] \quad = \quad C\eta_{\tau_1} ; \lceil \lfloor \mathtt{id}_{\tau} \rfloor ; \mathtt{coexec}_{M\tau} \rceil$$

Analogously, for any monad M on a category $\mathbf{D}$ and comonad C on the same category $\mathbf{D}$ along with a distributive law σ of M over C (so that $K^{\sigma}_{C,M}$ is a well-defined category), $K^{M}_{C,M}$, $K^{\sigma}_{C,M}$, and $K^{C}_{C,M}$ are all isomorphic.

The latter half of Theorem 6 is given by Power and Watanabe [2002], and with the former half we directly connect this to doubly-effectful languages. Theorem 6 implies that the layerings are strictly more expressive than distributive laws. In particular, any monad and comonad whose monad-prioritizing and comonad-prioritizing layerings differ semantically cannot have a distributive law, including the monad and comonad we define in the next section to formalize our insight about strictness and laziness. Furthermore, the layerings apply to any situation where a distributive law applies and arrive at the same result. Thus, it is never necessary to identify a distributive law to give semantics to a doubly-effectful language, though it still can be useful.

This also provides the proof that linear logic has no distributive laws. If there were, then there would not be a meaningful difference between Schellinx's [1994] two translations of classical logic.

2.5 Languages with Multiple Inputs and Multiple Outputs

Experts may be interested in the categorical properties that ! and ? exhibit that enable the layerings in Sections 2.4.2 and 2.4.2 to generalize to languages with multiple inputs and multiple outputs. As is already known, the properties of being a monad and a comonad are not sufficient [Jones and Hudak, 1993; Moggi, 1989; Uustalu and Vene, 2008]. The rest of this section addresses these experts. As such, we will not define all of the terms in this section, instead relying on the expertise of the reader.

First, for multiple inputs, consider the well-established notion of strength for monads. Strength enables a monad M to generalize to multiple inputs via the

following rule:

$$\frac{\Gamma, \tau_1 \vdash M\tau_2}{\Gamma, M\tau_1 \vdash M\tau_2}$$

However, to generalize the layerings of M with a comonad C we only need the monad to admit that rule restricted to C:

$$\frac{C\Gamma, \tau_1 \vdash M\tau_2}{C\Gamma, M\tau_1 \vdash M\tau_2}$$

This can be achieved with by requiring M to be strong *relative to* C [Blute et al., 1996]. This weakening of strength is important because ? is not strong, but it is strong relative to !. In traditional categorical terms, strength relative to C is a natural transformation of type $M\tau_1 \otimes C\tau_2 \to M(\tau_1 \otimes C\tau_2)$ satisfying adaptations of the traditional strength laws, as detailed by Blute et al. [1996]. Note that traditional strength from a non-linear setting translates to strength relative to ! in a linear setting. Thus, any traditional strong monad can be layered with !, even in multi-input languages, to develop a strict and lazy semantics for the effect represented by that monad.

As for the comonad C, being comonadic only allows `cobind` to apply when there is a single input. In order to admit the multiple-input rule

$$\frac{C\Gamma \vdash \tau}{C\Gamma \vdash C\tau}$$

C must furthermore be lax monoidal with respect to $\otimes$, and it must satisfy the laws for a (symmetric) lax monoidal comonad Uustalu and Vene [2008]. These two properties are sufficient for generalizing our layerings to languages with multiple inputs and a single output.

Second, for multiple outputs, M and C must satisfy properties dual to those

above. In other words, C must admit the rule

$$\frac{C\tau_1 \vdash \tau_2, M\Delta}{C\tau_1 \vdash C\tau_2, M\Delta}$$

This can be achieved by requiring C to be strong *relative to* M [Blute et al., 1996]. In traditional categorical terms, strength relative to M is a natural transformation of type $C(\tau_1 \parr M\tau_2) \to C\tau_1 \parr M\tau_2$ satisfying adaptations of the traditional strength laws, as detailed by Blute et al.. Furthermore, in order for the monad M to admit the multiple-output rule

$$\frac{\tau \vdash M\Delta}{M\tau \vdash M\Delta}$$

M must be colax monoidal with respect to $\parr$, and it must satisfy the laws for a (symmetric) colax monoidal monad. These properties are sufficient for generalizing our layerings to languages with multiple inputs and/or multiple outputs.

2.6 Giving Semantics to Choice in Comp

In Section 2.4, we considered effectful languages in which the effects could be thunked into a monad and/or comonad. This allowed us to develop metatheories about when effects are necessarily monadic or comonadic, and about when distributive laws can or cannot exist. However, not all effectful languages are expressive enough to express their effects internally. Comp is an example. Comp has consumer choice and producer choice, but only one type $\mathbb{N}$, so it is not able to represent those effects as comonads and monads within its own type system.

This problem is addressed by formalizing an effectful language $\mathcal{L}$ in terms of another language $\mathcal{P}$. This other language is typically pure in the sense that it may

exhibit more desirable properties like confluence or termination. Furthermore, this other language is typically more expressive, especially with respect to its type system. In particular, the language's type system is capable of encoding various comonads and monads which can be used to formalize the semantics of various effects.

So to give a categorical semantics for a doubly-effectful language $\mathcal{L}$, one picks a "pure" language $\mathcal{P}$ along with a comonad C and a monad M on $\mathcal{P}$. Then one decides upon a Kleisli-like category that captures the desired interaction of effects. In particular, $K^M_{C,M}$ from Section 2.4 captures the monad-prioritizing (strict) semantics, whereas $K^C_{C,M}$ captures the comonad-prioritizing (lazy) semantics. The choice of category provides the semantics for sequentially composing programs in the effectful language. Thus, after making this choice, one simply has to provide translations of the primitive operations from the effectful language into this category, and the remainder of the semantics will be derived from the categorical structure. Furthermore, because $K^M_{C,M}$ and $K^C_{C,M}$ have already been proven to be well-formed categories, the derived semantics is guaranteed to sequentially compose effectful programs in a manner that is associative and respects identities. Technically these constructions only apply to single-input, single-output languages, but they are easy to extend to multiple inputs and/or multiple outputs, the formalization of which is in Section 2.5.

In the following sections we explicitly construct translations for Comp—one for strict semantics and one for lazy semantics. But to do so, we first need a pure language with a comonad and monad capable of representing consumer choice and producer choice. We develop such a language next.

Channels	$x, y, z, \ldots$		
Constants	c	$::=$	$0 \mid 1 \mid \ldots$
Processes	ρ, ν	$::=$	$\varnothing$
		$\mid$	$\rho_1 \parallel \rho_2$
		$\mid$	$y.\mathsf{init}(c)$
		$\mid$	$x \rightleftharpoons y$
		$\mid$	$y.\mathsf{send}()$
		$\mid$	$y.\mathsf{send}(x)$
		$\mid$	$\mathsf{handle}_{?x}\{\rho\}$
		$\mid$	$x.\mathsf{req}()$
		$\mid$	$x.\mathsf{req}(y)$
		$\mid$	$x.\mathsf{req}(!y_1, !y_2)$
		$\mid$	$\mathsf{supply}_{!y}\{\rho\}$
Types	τ	$::=$	$\mathbb{N} \mid\ !\tau \mid\ ?\tau$
Contexts	Γ, Δ, Ξ	$::=$	$x : \tau, \ldots$
			(no repeats)
			(unordered)

with the following syntactic identifications

$$\rho_1 \parallel (\rho_2 \parallel \rho_3) \equiv (\rho_1 \parallel \rho_2) \parallel \rho_3$$
$$\rho_1 \parallel \rho_2 \equiv \rho_2 \parallel \rho_1$$
$$\varnothing \parallel \rho \equiv \rho$$

Figure 2.12: Proc Syntax

$$\frac{}{\vdash \varnothing \dashv}$$

$$\frac{\Gamma \vdash \rho_1 \dashv \Delta, \Xi \qquad \Gamma', \Xi \vdash \rho_2 \dashv \Delta'}{\Gamma, \Gamma' \vdash \rho_1 \parallel \rho_2 \dashv \Delta, \Delta'}$$

$$\frac{}{\vdash y.\mathsf{init}(c) \dashv y : \mathbb{N}}$$

$$\frac{}{x : \tau \vdash x \rightleftharpoons y \dashv y : \tau}$$

$$\frac{}{\vdash y.\mathsf{send}() \dashv y : ?\tau}$$

$$\frac{}{x : \tau \vdash y.\mathsf{send}(x) \dashv y : ?\tau}$$

$$\frac{!\Gamma, x : \tau_1 \vdash \rho \dashv y : ?\tau_2}{!\Gamma, x : ?\tau_1 \vdash \mathsf{handle}_{?x}\{\rho\} \dashv y : ?\tau_2}$$

$$\frac{}{x : !\tau \vdash x.\mathsf{req}() \dashv}$$

$$\frac{}{x : !\tau \vdash x.\mathsf{req}(y) \dashv y : \tau}$$

$$\frac{}{x : !\tau \vdash x.\mathsf{req}(!y_1, !y_2) \dashv y_1 : !\tau, y_2 : !\tau}$$

$$\frac{!\Gamma \vdash \rho \dashv y : \tau}{!\Gamma \vdash \mathsf{supply}_{!y}\{\rho\} \dashv y : !\tau}$$

Figure 2.13: Proc Typing Rules

2.6.1 A Language without Consumer or Producer Choice

Here we develop a language with neither consumer choice nor producer choice, instead using a comonad ! and a monad ? to capture those effects. By translating Comp into this language we force ourselves to explicitly commit to a particular interaction between these effects.

As suggested in Section 2.3, we use a computational model of linear logic that makes the effects of Comp explicit. There are several such models to choose from. Perhaps the most common is linear λ-calculus [Wadler, 1990]. However, linear λ-calculus is based on intuitionistic linear logic, so it does not model the ? ex-

ponential. The other obvious choice is π-calculus, which has been shown to be a computational model for full classical linear logic [Abramsky, 1994; Beffara, 2005; Bellin and Scott, 1994; DeYoung et al., 2012; Milner et al., 1992; Wadler, 2012]. However, π-calculus was originally designed to study mobile processes [Milner et al., 1992], and the constructions used to model terms of linear logic are complicated. We thus choose to build our own calculus, Proc, based directly on classical linear logic, and specialized to our application.[6] Proc is essentially a specialized fragment of π-calculus, and so our translations of Comp into Proc also translate Comp into π-calculus. However, Proc, being more suited to our setting, has a pedagogical advantage.

Syntax and Typing Rules

Proc is a multiple-input, multiple-output language based on parallel processes communicating through named channels, like π-calculus. Its syntax is presented in Figure 2.12. In this figure, we consistently use x to refer to "input" channels, and use y to refer to "output" channels, as an aid to the reader. For example, $y.\mathsf{send}(x)$ sends the input channel x onto the output channel y. Note that, in order to simplify our presentations throughout the paper, we treat $\parallel$ as operating on multisets. Thus we treat $\rho_1 \parallel \rho_2$ as being syntactically identical to $\rho_2 \parallel \rho_1$, as formalized among other syntactic identities by $\equiv$ in Figure 2.12.

The typing rules for Proc are presented in Figure 2.13, once again using x for inputs and y for outputs. We write $\Gamma \vdash \rho \dashv \Delta$, where Γ and Δ are disjoint contexts, to mean that ρ consumes the channels in Γ and produces the channels

[6] In particular, we take advantage of the lack of contraction for ? to simplify our syntax and reduce the number of DISTRIBUTE rules in our semantics. As a result, a judgment $\Gamma \vdash \rho \dashv \Delta$ corresponds to $\bigotimes \Gamma \vdash \bigotimes \Delta$ rather than $\bigotimes \Gamma \vdash \parr \Delta$.

in Δ. Intuitively, we can think of this as saying that ρ will receive messages from the channels in Γ and send messages on the channels in Δ. Note, though, that this is just an intuition. As with many process calculi, channels in Proc are bidirectional, so a consumer of a channel can just as well send messages on that channel, and likewise a producer can receive messages.

The three simplest processes are $y.\mathsf{init}(c)$, $x \rightleftharpoons y$, and $\varnothing$. The process $y.\mathsf{init}(c)$ simply produces the channel y by providing the constant c, representing variable initialization in Comp. The process $x \rightleftharpoons y$ is viewed as consuming x and producing y, but it simply forwards messages received on either channel to the other, essentially unifying the two channels. The process $\varnothing$ does nothing and has no inputs or outputs.

Parallel composition $\|$ is the fundamental way to compose processes in Proc. Any channels that two parallel processes have in common are implicitly connected together. Recall also that $\|$ is commutative, so the typing rule does not force a left-to-right order. To prevent potential resulting ambiguities, we make a whole-process assumption that every channel occurs at most once across all Ξ contexts in the typing proof for a Proc process. This is analogous to the whole-program assumption that every variable is assigned to at most once in Comp.

There are two ways to produce a $?\tau$ channel y, both of which model possible producer choices in Comp. We model choosing to produce no output with the syntax $y.\mathsf{send}()$, which sends an empty message on y. To model choosing to produce one output, we use the syntax $y.\mathsf{send}(x)$, which sends the channel x on y. The process that listens to y can then use x to get the input it needs to run.

To consume a $?\tau$ channel x, one uses the process $\mathsf{handle}_{?x}\{\rho\}$. This process

uses ρ to handle the messages sent on x, unpacking their content before forwarding the messages on x to ρ. Note that the producer of x will send either one or zero messages on x, so ρ will be run either once or not at all, explicitly representing Comp's producer choice of quantity.

Channels of type $!\tau$ are conceptually dual to $?\tau$ channels. Whereas producers of $?\tau$ send channels to be received by handlers, consumers of $!\tau$ request channels from suppliers. The process $x.\mathsf{req}()$ requests nothing from x, which models a Comp program choosing not to use its input. The process $x.\mathsf{req}(y)$ requests a channel y from x, which models a Comp program choosing to use its input. But there is a slight asymmetry: a Comp program can only choose to produce zero or one outputs, but can choose to consume an input zero, one, *or multiple* times. Consequently, $x.\mathsf{req}(!y_1, !y_2)$ requests two channels from x, and furthermore these channels must be able to process additional requests so that a consumer can use an input as many times as it wants.

To produce a $!\tau$ channel y, one uses the process $\mathsf{supply}_{!y}\{\rho\}$. This process repeatedly uses ρ to supply for the requests made by the consumer of y. Such requests can effectively be made an arbitrary number of times, explicitly representing Comp's consumer choice of quantity.

Semantics

We formalize the behavior of Proc using the reduction rules in Figure 2.14. These reduction rules were developed from cut elimination in classical linear logic. In fact, it is relatively simple to show that this is equivalent to a fragment of the π-calculus model of linear logic developed by Beffara [2005]. As a consequence, they enjoy the properties of progress, preservation, confluence, and termination; we prove as

$$\textsc{Parallel}\quad \frac{\rho_1 \to \rho_1'}{\rho_1 \parallel \rho_2 \to \rho_1' \parallel \rho_2}$$

$$\textsc{Context}\quad \frac{\rho \to \rho'}{\mathsf{handle}_{?x}\{\rho\} \to \mathsf{handle}_{?x}\{\rho'\}} \qquad \frac{\rho \to \rho'}{\mathsf{supply}_{!z}\{\rho\} \to \mathsf{supply}_{!z}\{\rho'\}}$$

$$\textsc{Identity}\quad \frac{y \in \mathsf{consumed}(\rho)}{x \rightleftharpoons y \parallel \rho \to \rho[y \mapsto x]} \qquad \frac{y \in \mathsf{produced}(\rho)}{\rho \parallel y \rightleftharpoons z \to \rho[y \mapsto z]}$$

$$\textsc{Fail}\quad \frac{\begin{array}{c}\rho_x = \{x.\mathsf{req}() \mid x \in \mathsf{consumed}(\rho) \wedge x \neq y\} \\ \rho_z = \{z.\mathsf{send}() \mid z \in \mathsf{produced}(\rho)\}\end{array}}{y.\mathsf{send}() \parallel \mathsf{handle}_{?y}\{\rho\} \to \rho_x \parallel \rho_z}$$

$$\textsc{Succeed}\quad y.\mathsf{send}(x) \parallel \mathsf{handle}_{?y}\{\rho\} \to \rho[y \mapsto x]$$

$$\textsc{Drop}\quad \mathsf{supply}_{!y}\{\rho\} \parallel y.\mathsf{req}() \to \{x.\mathsf{req}() \mid x \in \mathsf{consumed}(\rho)\}$$

$$\textsc{Take}\quad \mathsf{supply}_{!y}\{\rho\} \parallel y.\mathsf{req}(z) \to \rho[y \mapsto z]$$

$$\textsc{Clone}\quad \frac{\begin{array}{c}\text{for each } x \in \mathsf{consumed}(\rho), \text{ the channels } y_1^x \text{ and } y_2^x \text{ are fresh} \\ \rho_x = \{x.\mathsf{req}(!y_1^x, !y_2^x) \mid x \in \mathsf{consumed}(\rho)\} \\ \rho_{z_1} = \mathsf{supply}_{!z_1}\{\rho\,[y \mapsto z_1,\, x \mapsto y_1^x \mid x \in \mathsf{consumed}(\rho)]\} \\ \rho_{z_2} = \mathsf{supply}_{!z_2}\{\rho\,[y \mapsto z_2,\, x \mapsto y_2^x \mid x \in \mathsf{consumed}(\rho)]\}\end{array}}{\mathsf{supply}_{!y}\{\rho\} \parallel y.\mathsf{req}(!z_1, !z_2) \to \rho_x \parallel \rho_{z_1} \parallel \rho_{z_2}}$$

$$\textsc{Distribute}\quad \frac{y \in \mathsf{consumed}(\rho_2)}{\mathsf{supply}_{!y}\{\rho_1\} \parallel \mathsf{supply}_{!z}\{\rho_2\} \to \mathsf{supply}_{!z}\{\mathsf{supply}_{!y}\{\rho_1\} \parallel \rho_2\}}$$

$$\frac{y \in \mathsf{consumed}(\rho_2)}{\mathsf{supply}_{!y}\{\rho_1\} \parallel \mathsf{handle}_{?x}\{\rho_2\} \to \mathsf{handle}_{?x}\{\mathsf{supply}_{!y}\{\rho_1\} \parallel \rho_2\}}$$

$$\frac{y \in \mathsf{produced}(\rho_1)}{\mathsf{handle}_{?x}\{\rho_1\} \parallel \mathsf{handle}_{?y}\{\rho_2\} \to \mathsf{handle}_{?x}\{\rho_1 \parallel \mathsf{handle}_{?y}\{\rho_2\}\}}$$

Figure 2.14: Proc Reduction Rules

much in Appendix A. Thus, even though Proc is a highly parallel calculus, every Proc process will compute to the same value no matter how it is reduced. Proc is designed so that we can consider two processes semantically equivalent precisely when they reduce to syntactically identical normal forms, modulo renaming of intermediate channels.

To assist the reader, our presentation of the reduction rules in Figure 2.14 consistently conform to a few conventions. We continue to use x to refer to input channels, but now we use z to refer to output channels. As for y, here we use it to refer to intermediate channels. Most rules have a producer of y occurring in parallel with a consumer of y, and the reduction often then eliminates y from the process altogether. Hence, y is the cutpoint of the reduction. Meanwhile, the reduced term will always have the same input channels x and output channels z as the original, as is consistent with the linear nature of the calculus. Lastly, although $\|$ is commutative, for convenience we also present producers of intermediate channels to the left of $\|$, and consumers to the right, providing a more familiar left-to-right reading of the processes.

The Parallel and Context rules together say that reduction can occur anywhere within the process, provided there is an appropriate opportunity for reduction. Note that the Parallel rule does not restrict reduction to only the left-hand side of $\|$ because $\|$ is syntactically commutative.

The Identity rules say that $x \rightleftharpoons y$ is essentially the identity process. These rules refer to the properties $\mathsf{produced}(\rho)$ and $\mathsf{consumed}(\rho)$, which are formalized in Appendix A. Informally, a channel is produced by ρ if it is an input of ρ according to the type of ρ, and a channel is consumed by ρ if it is an output of ρ according to the type of ρ. The identity rules check these properties in order to ensure that the

channel being substituted is internal to the system rather than an exposed input or output of the system.

The FAIL rule defines what happens when a handler $\mathsf{handle}_{?y}\{\rho\}$ is sent an empty message. This indicates the handler is not needed. Consequently, ρ is eliminated from the process. Furthermore, empty messages are dispatched (by ρ_x and ρ_z) on the other channels that ρ would have consumed or produced. This lets the other users of those channels know that they will not be needed. Note that we use comprehension notation to define ρ_x and ρ_z, taking advantage of the fact that the parallel composition operator $\|$ operates on multisets of processes.

The SUCCEED rule defines what happens when a handler $\mathsf{handle}_{?y}\{\rho\}$ is sent a (single) channel x. This indicates that the handler should be executed (once) using x as its input in place of y. As such, this just reduces to ρ with its y input substituted with x.

The DROP and TAKE rules are dual to the SUCCEED and FAIL rules, and they look very similar. The only differences besides duality are due to the fact that handlers and suppliers can consume many input channels but produce only one output channel, a restriction that simplifies our presentation while still being sufficient for representing Comp.

The CLONE rule defines what happens when a supplier is given a request for two reusable channels. This reduction works by duplicating the supplier. However, the supplier may be consuming a variety of channels. Consequently, requests must be dispatched (by ρ_x) to each of these channels so that they are duplicated as well, with the two new suppliers (ρ_{z_1} and ρ_{z_2}) being connected to the appropriate duplicates. Thus the CLONE rule is the only reduction that introduces new intermediate

channels, although the original intermediate channel y is still eliminated.

Finally, the DISTRIBUTE rules allow suppliers and handlers to be pulled into other suppliers and handlers when communicating on appropriate channels of the contained process. This is important because reduction can proceed inside suppliers and handlers. In particular, the DISTRIBUTE rules are necessary for proving that ! and ? satisfy the (co)monad laws, enabling them to represent the consumer and producer effects of Comp. Consequently we can apply the layering techniques from Section 2.4 to give Comp a categorical semantics using Proc.

2.6.2 Layering Effects

Now that we have constructed Proc and Comp, we want to show that the comonad and monad of Proc actually capture the effects of Comp. To do that, we translate the structural rules of Comp into the structural rules of Proc, which are only available when using ! and ?. Since Comp is analogous to classical logic and Proc to linear logic, this is similar to the challenge of embedding classical logic into linear logic. Indeed, as before, we cannot give a distributive law between ! and ?, so we must instead use the layerings.

This time, however, we also have to consider terms and β reduction. In particular, we need to show that we translate Comp programs to Proc processes with the same semantics. We give the translations in Figures 2.15 and 2.16.

The translation in Figure 2.15 is the monad-prioritizing layering $!\Gamma \vdash ?!\Delta$, derived from using the Kleisli-like category $K^M_{C,M}$. Note that it composes processes using $\mathsf{handle}_{?x}\{\rho\}$. Handle implements bind for the ? monad in Proc. By using bind, this translation corresponds to the strict semantics of Comp, always prop-

$$\vdash x \coloneqq c \dashv x : \mathbb{N} \;\rightsquigarrow_s\; \vdash \mathsf{supply}_{!x'}\{x'.\mathsf{init}(c)\} \parallel x.\mathsf{send}(x') \dashv x : ?!\mathbb{N} \quad (x' \text{ fresh})$$

$$y : t \vdash x \coloneqq y \dashv x : t \;\rightsquigarrow_s\; y : !t \vdash x.\mathsf{send}(y) \dashv x : ?!t$$

$$\vdash x \coloneqq \mathbf{error} \dashv x : t \;\rightsquigarrow_s\; \vdash x.\mathsf{send}() \dashv x : ?!t$$

$$\frac{\Gamma \vdash p_1 \dashv x : t_1 \;\rightsquigarrow_s\; !\Gamma \vdash \rho_1 \dashv x : ?!t_1 \qquad \Gamma', x : t_1 \vdash p_2 \dashv y : t_2 \;\rightsquigarrow_s\; !\Gamma', x : !t_1 \vdash \rho_2 \dashv y : ?!t_2}{\Gamma, \Gamma' \vdash p_1;\; p_2 \dashv y : t_2 \;\rightsquigarrow_s\; !\Gamma, !\Gamma' \vdash \rho_1 \parallel \mathsf{handle}_{?x}\{\rho_2\} \dashv y : ?!t_2}$$

$$\frac{\Gamma \vdash p \dashv x : t_1 \;\rightsquigarrow_s\; !\Gamma \vdash \rho \dashv x : ?!t_1}{\Gamma, y : t_2 \vdash p \dashv x : t_1 \;\rightsquigarrow_s\; !\Gamma, y : !t_2 \vdash y.\mathsf{req}() \parallel \rho \dashv x : ?!t_1}$$

$$\frac{\Gamma, x_1 : t_1, x_2 : t_1 \vdash p \dashv y : t_2 \;\rightsquigarrow_s\; !\Gamma, x_1 : !t_1, x_2 : !t_1 \vdash \rho \dashv y : ?!t_2}{\Gamma, x : t_1 \vdash p \dashv y : t_2 \;\rightsquigarrow_s\; !\Gamma, x : !t_1 \vdash x.\mathsf{req}(!x_1, !x_2) \parallel \rho \dashv y : ?!t_2}$$

Figure 2.15: Strict Translation

$$\vdash x \coloneqq c \dashv x : \mathbb{N} \;\rightsquigarrow_\ell\; \vdash x'.\mathsf{init}(c) \parallel x'.\mathsf{send}(x) \dashv x : ?\mathbb{N} \quad (x' \text{ fresh})$$

$$y : t \vdash x \coloneqq y \dashv x : t \;\rightsquigarrow_\ell\; y : !?t \vdash y.\mathsf{req}(x) \dashv x : ?t$$

$$\vdash x \coloneqq \mathbf{error} \dashv x : t \;\rightsquigarrow_\ell\; \vdash x.\mathsf{send}() \dashv x : ?t$$

$$\frac{\Gamma \vdash p_1 \dashv x : t_1 \;\rightsquigarrow_\ell\; !?\Gamma \vdash \rho_1 \dashv x : ?t_1 \qquad \Gamma', x : t_1 \vdash p_2 \dashv y : t_2 \;\rightsquigarrow_\ell\; !?\Gamma', x : !?t_1 \vdash \rho_2 \dashv y : ?t_2}{\Gamma, \Gamma' \vdash p_1;\; p_2 \dashv y : t_2 \;\rightsquigarrow_\ell\; !?\Gamma, !?\Gamma' \vdash \mathsf{supply}_{!x}\{\rho_1\} \parallel \rho_2 \dashv y : ?t_2}$$

$$\frac{\Gamma \vdash p \dashv x : t_1 \;\rightsquigarrow_\ell\; !?\Gamma \vdash \rho \dashv x : ?t_1}{\Gamma, y : t_2 \vdash p \dashv x : t_1 \;\rightsquigarrow_\ell\; !?\Gamma, y : !?t_2 \vdash y.\mathsf{req}() \parallel \rho \dashv x : ?t_1}$$

$$\frac{\Gamma, x_1 : t_1, x_2 : t_1 \vdash p \dashv y : t_2 \;\rightsquigarrow_\ell\; !?\Gamma, x_1 : !?t_1, x_2 : !?t_1 \vdash \rho \dashv y : ?t_2}{\Gamma, x : t_1 \vdash p \dashv y : t_2 \;\rightsquigarrow_\ell\; !?\Gamma, x : !?t_1 \vdash x.\mathsf{req}(!x_1, !x_2) \parallel \rho \dashv y : ?t_2}$$

Figure 2.16: Lazy Translation

agating errors forward through the process regardless of whether the values are needed. For this reason, we also refer to this translation as the strict translation. To see this strict behavior, consider the strict translation of our example Comp program $x \coloneqq 3;\ y \coloneqq \mathbf{error};\ z \coloneqq x$:

$$\mathsf{supply}_{!x'}\{x'.\mathsf{init}(3)\} \parallel x.\mathsf{send}(x') \parallel \mathsf{handle}_{?x}\{y.\mathsf{send}() \parallel \mathsf{handle}_{?y}\{y.\mathsf{req}() \parallel z.\mathsf{send}(x)\}\}$$

This can reduce on either x or y. Since Proc is confluent (as proved in Appendix A), we can reduce on either channel and get the same result. Choose x. The first thing this process does is reduce $x.\mathsf{send}(x')$ and $\mathsf{handle}_{?x}\{\rho\}$, reducing to $\rho[x \mapsto x']$. We then reduce on y, reducing $y.\mathsf{send}()$ and $\mathsf{handle}_{?y}\{\rho\}$ to $x'.\mathsf{req}()$ and $z.\mathsf{send}()$, thereby propagating the error. Then the supplier of x' receives an empty message, which reduces to nothing, leaving just $z.\mathsf{send}()$. This is what we would get if we reduced the Comp program using the strict semantics and then translated the result to Proc.

The translation in Figure 2.16 corresponds to the comonad-prioritizing layering $!?\Gamma \vdash ?\Delta$, derived from using the Kleisli-like category $K^{C}_{C,M}$. It composes processes using $\mathsf{supply}_{!x}\{\rho\}$. Supply implements cobind for the ! comonad in Proc. For this reason, this translation corresponds to the lazy semantics of Comp, always propagating whether values are needed before evaluating them.

With the lazy translation, our example Comp program $x \coloneqq 3;\ y \coloneqq \mathbf{error};\ z \coloneqq x$ becomes

$$\mathsf{supply}_{!x}\{x'.\mathsf{init}(3) \parallel x.\mathsf{send}(x')\} \parallel \mathsf{supply}_{!y}\{y.\mathsf{send}()\} \parallel y.\mathsf{req}() \parallel x.\mathsf{req}(z)$$

Since this can reduce on either x or y and either way will arrive at the same result, assume that y is chosen. Then, the supplier of y receives an empty request, which reduces to nothing. Note that this removes the only empty send in the process,

thereby ignoring the error. Next, reducing on x results in $x'.\mathsf{init}(3) \parallel z.\mathsf{send}(x')$, which is irreducible and is what we would get (modulo renaming x') if we reduced the Comp program using the lazy semantics and then translated the result to Proc.

The following theorem shows that choosing strictness versus laziness is always the same as choosing to prioritize producer choice versus consumer choice. This generalizes to *any* strong monad for a non-linear language, as explained in Section 2.5. This means our comonad-prioritizing layering defines a lazy semantics for every strong monad. Furthermore, whenever the strict and lazy semantics for a strong monad differ, Theorem 6 proves that there cannot be a distributive law of that monad over !.

Theorem 7 (Semantic Preservation)**.** *Suppose that a Comp program p translates strictly to a Proc process ρ, meaning that $\Gamma \vdash p \dashv x : t \rightsquigarrow_s\ !\Gamma \vdash \rho \dashv x : ?!t$, and a Comp program $x := e$ translates strictly to a Proc process ν, meaning that $\Gamma \vdash x := e \dashv x : t \rightsquigarrow_s\ !\Gamma \vdash \nu \dashv x : ?!t$. Then p reduces strictly to $x := e$, meaning $p \to_s^* x := e$, if and only if ρ reduces to ν, meaning $\rho \to^* \nu$. The theorem statement also holds for laziness, meaning with $\rightsquigarrow_\ell$ in place of $\rightsquigarrow_s$ and $\to_\ell^*$ in place of $\to_s^*$.*

Proof. First, we prove that if p strictly reduces to $x := e$, then ρ reduces to ν. Since Proc is canonicalizing (as proved in Appendix A), we need only show that ρ *can* reduce to ν.

We can do this by induction on $\Gamma \vdash p \dashv x : t$. The only interesting case is

$$\frac{\Gamma_1 \vdash p_1 \dashv x : t \ \rightsquigarrow_s\ \rho_1 \qquad \Gamma_2, x : t \vdash p_2 \dashv y : t' \ \rightsquigarrow_s\ \rho_2}{\Gamma_1, \Gamma_2 \vdash p_1; p_2 \dashv y : t' \ \rightsquigarrow_s\ \rho_1 \parallel \mathsf{handle}_{?x}\{\rho_2\}}$$

By induction, if p_1 reduces strictly to $x := e_1$ and p_2 reduces strictly to $y := e_2$

then $\rho_1 \to^* \nu_1$ and $\rho_2 \to^* \nu_2$. We then consider the cases of e_1, here ignoring the drops of unused inputs for brevity.

If e_1 is c, then $p_1; p_2$ will eventually strictly reduce to $p_2[x \mapsto c]$. On the Proc side, ν_1 will be $x'.\mathsf{init}(c) \parallel x'.\mathsf{send}(x)$, which will match with the handler. This will cause ρ_2 to run normally with the variable x mapped to x', which is initialized to the value c. This is the translation of $p_2[x \mapsto c]$. A similar argument holds if e_1 is some variable z.

If e_1 is **error**, then $p_1; p_2$ will eventually strictly reduce to an **error**. On the Proc side, ν_1 will be $x.\mathsf{send}()$, which will match with the handler and altogether reduce to $y.\mathsf{send}()$, because y is necessarily $\mathsf{produced}$ by ρ_2. This is the translation of $y \coloneqq \mathbf{error}$.

A similar proof holds for the lazy semantics except one considers the cases of e_2 instead of e_1.

Lastly, we prove the reverse direction of the if and only if: that if ρ reduces to ν, then p strictly reduces to $x \coloneqq e$, and similarly for the lazy semantics. Because strict Comp is canonicalizing, p must reduce to $x \coloneqq e'$ for some e', and $x \coloneqq e'$ translates to some ν'. Then, the first half of the theorem implies that ρ reduces to ν'. By the definition of translation of assignments, ν and ν' are easily shown to be irreducible. Thus, ρ reduces to two irreducible processes, ν and ν', which must then be equal since Proc is canonicalizing. This means $x \coloneqq e$ and $x \coloneqq e'$ translate to the same Proc process, which is easily shown to imply that $e = e'$. $\square$

CHAPTER 3

BELIEF SEMANTICS OF AUTHORIZATION LOGIC

Authorization logics are used in computer security to reason about whether *principals*—computer or human agents—are permitted to take actions in computer systems. The distinguishing feature of authorization logics is their use of a "says" connective: intuitively, if principal p believes that formula φ holds, then formula p says φ holds. Access control decisions can then be made by reasoning about (i) the beliefs of principals, (ii) how those beliefs can be combined to derive logical consequences, and (iii) whether those consequences entail *guard formulae*, which must hold for actions to be permitted.

Many systems that employ authorization logics have been proposed (see Abadi [2003]; Chapin et al. [2008] for surveys; see also Appel and Felten [1999]; Bauer et al. [2005]; Becker and Sewell [2004]; Becker et al. [2010]; Cederquist et al. [2007]; Cirillo et al. [2007]; DeTreville [2002]; Fournet et al. [2005]; Gurevich and Neeman [2008]; Jia et al. [2008]; Jim [2001]; Lampson et al. [1991]; Lesniewski-Laas et al. [2007]; Li et al. [2000, 2002]; Pimlott and Kiselyov [2006]; Polakow and Skalka [2006]; Sirer et al. [2011]; Wobber et al. [1994]), but few authorization logics have been given formal models [Abadi et al., 1993; Garg, 2008; Garg and Abadi, 2008; Genovese et al., 2012; Howell, 2000]. Though models might not be immediately necessary to deploy authorization logics in real systems, they yield insight into the meaning of formulae and enable us to prove soundness of a proof system—which might require proof rules and axioms to be corrected, if there are any lurking errors in the proof system. Moreover, certain algorithms—e.g., for exploring the consequences of an authorization policy—use a formal semantics directly [Genovese et al., 2012].

For the sake of security, it is worthwhile to carry out such soundness proofs. Given only a proof system, we must trust that the proof system is correct. But given a proof system and a soundness proof, which shows that any provable formula is semantically valid, we now have evidence that the proof system is correct and hence trustworthy. The soundness proof thus relocates trust from the proof system to the proof itself as well as to the model theory which ideally offers more intuition about formulae than the proof system itself.

Semantics of authorization logics are usually based on *possible worlds*, as used by Kripke [1963]. *Kripke semantics* posit an indexed *accessibility relation* on possible worlds. If at world w, principal p considers world w' to be possible, then (w, w') is in p's accessibility relation. We denote this as $w \leq_p w'$. Authorization logics sometimes use Kripke semantics to give meaning to the says connective: semantically, p says φ holds in a world w if and only if for all worlds w' such that $w \leq_p w'$, formula φ holds in world w'. Hence a principal says φ if and only if φ holds in all worlds the principal considers possible.[1]

The use of Kripke semantics in authorization logic thus requires installation of possible worlds and accessibility relations into the semantics, solely to give meaning to says. That's useful for studying properties of logics and for building decision procedures. But, unfortunately, it doesn't seem to correspond to how principals reason in real-world systems. Rather than explicitly considering possible worlds and relations between them, principals typically begin with some set of base formulae they believe to hold—perhaps because they have received digitally signed messages encoding those formulae, or perhaps because they invoke system calls that return information—then proceed to reason from those formulae. So

[1]The says connective is, therefore, closely related to the modal necessity operator $\Box$ [Hughes and Cresswell, 1996] and the epistemic knowledge operator K [Fagin et al., 1995].

could we instead stipulate that each principal p have a set of beliefs $\omega(p)$, called the *worldview* of p, such that p says φ holds if and only if $\varphi \in \omega(p)$? That is, a principal says φ if and only if φ is in the worldview[2] of the principal?

This chapter answers that question in the affirmative. We give two classes of models for an authorization logic: one class (Section 3.2) of Kripke models, the other (Section 3.1) introduces *belief models*, which employ worldviews to interpret says.[3] We show (Section 3.3) that belief models subsume Kripke models, in the sense that every Kripke model can be transformed into a belief model. A formula is valid in the Kripke model if and only if it is also valid in the belief model. As a result, our authorization logic can now eliminate the technical machinery of Kripke semantics and instead use *belief semantics*. This semantics potentially increases the trustworthiness of an authorization system because the semantics is closer to how principals reason in real systems.

The particular logical system we introduce in this paper is FOCAL, First-Order Constructive Authorization Logic. FOCAL extends a well-known authorization logic, cut-down dependency core calculus (CDD) [Abadi, 2006], from a propositional language to a language with first-order functions and relations on system state. Functions and relations are essential for reasoning about authorization in a real operating system—as exemplified in Nexus Authorization Logic (NAL) [Schneider et al., 2011], of which FOCAL is a fragment. FOCAL also simplifies NAL by eliminating second-order quantification.

Having given two semantics for FOCAL, we then turn to the problem of proving soundness. It turns out that the NAL proof system is unsound with respect to the

[2]Worldviews were first employed by NAL [Schneider et al., 2011], which pioneered an informal semantics based on them.

[3]Our belief models are an instance of the *syntactic* approach to modeling knowledge [Eberle, 1974; Fagin et al., 1995; Moore and Hendrix, 1979].

semantics presented here: NAL allows the derivation of a well-known formula (see Section 3.4.1) that our semantics deems invalid. In particular, our belief semantics demonstrated that if a logic is to be used in a distributed setting without globally-agreed upon state, then its proof system should not allow this well-known formula to be derived. So if NAL is to be used in such settings, its proof system needs to be corrected.

NAL extends CDD, which is also unsound with respect to our semantics. However, CDD has been proven sound with respect to a different semantics [Garg and Abadi, 2008]. This seeming discrepancy—sound vs. unsound—illuminates a previously unexplored difference between how NAL and CDD interpret says.

To achieve soundness for FOCAL, we develop a revised proof system whose key technical change is adopting localized hypotheses in the proof rules. In Section 3.4, we prove the soundness of our proof system with respect to both our belief and Kripke semantics. This result yields the first soundness proof with respect to belief semantics for an authorization logic.

Having relocated trust into the soundness proof, we then seek a means to increase the trustworthiness of that proof. Accordingly, we formalize the syntax, proof system, belief semantics, and Kripke semantics of FOCAL in the Coq proof assistant [The Coq development team, 2004], and we mechanize the proofs of soundness for both the belief semantics and the Kripke semantics. That mechanization relocates trust from our soundness proof to the Coq proof system, which is well-studied and is the basis of many other formalizations. The full Coq formalization contains 3,496 lines of code and required about four person-months for us, as Coq neophytes, to develop [Hirsch and Clarkson, 2013b].

This chapter thus advances the theory of computer security with the following novel contributions:

- the first formal belief semantics for authorization logic (Section 3.1),
- a proof of equivalence between that belief semantics and a corresponding Kripke semantics (Sections 3.2 and 3.3),
- a proof system that is sound with respect to belief and Kripke semantics (Section 3.4), and
- the first machine-checked proof of soundness for an authorization-logic proof system (Section 3.4).

3.1 Belief Semantics

FOCAL is a constructive, first-order, multimodal logic. The key features that distinguishes it as an authorization logic are the says and speaksfor connectives, invented by Lampson et al. [1991]. These are used to reason about authorization—for example, access control in a distributed system can be modeled in the following standard way:

Example 1. A *guard* implements access control for a printer p. To permit printing to p, the guard must be convinced that guard formula *PrintServer* says $printTo(p)$ holds, where *PrintServer* is the principal representing the server process. That formula means that *PrintServer* believes $printTo(p)$ holds. To grant printer access to user u, the print server can issue the statement u speaksfor *PrintServer*. That formula means anything u says, the *PrintServer* must also say. So if u says $printTo(p)$, then *PrintServer* says $printTo(p)$, which satisfies the guard formula and hence affords the user access to the printer.

$$
\begin{array}{lrcl}
\text{Terms} & \tau & ::= & x \mid f(\tau_1, \ldots, \tau_n) \\
\text{Formulae} & \varphi, \psi, \ldots & ::= & \mathsf{true} \mid \mathsf{false} \mid r(\tau_1, \ldots, \tau_n) \mid \tau_1 = \tau_2 \\
& & \mid & \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \neg\varphi \\
& & \mid & (\forall\, x : \varphi) \mid (\exists\, x : \varphi) \\
& & \mid & \tau\ \mathsf{says}\ \varphi \mid \tau_1\ \mathsf{speaksfor}\ \tau_2
\end{array}
$$

Figure 3.1: Syntax of FOCAL

Figure 3.1 gives the formal syntax of FOCAL. There are two syntactic classes, terms τ and formulae φ. Metavariable x ranges over first-order variables, f over first-order functions, and r over first-order relations.

Formulas of FOCAL do not permit monadic second-order universal quantification, unlike CDD and NAL. In NAL, that quantifier was used only to define false and speaksfor as syntactic sugar. FOCAL instead adds these as primitive connectives to the logic. FOCAL also defines $\neg\varphi$ as a primitive connective, but it could equivalently be defined as syntactic sugar for $\varphi \Rightarrow \mathsf{false}$.

Syntactically, FOCAL is thus CDD without second-order quantification, but with first-order terms and quantification and a primitive speaksfor connective. Likewise, FOCAL is NAL without second-order quantification, subprincipals, group principals, and restricted delegation, but with a primitive speaksfor connective.

3.1.1 Semantic models

The belief semantics of FOCAL is based on a combination of two standard semantic models—first-order models and constructive models—with worldviews, which are used to interpret says and speaksfor. To our knowledge, this semantics is new in the study of authorization logics. Our presentation mostly follows the semantics

of intuitionistic predicate calculus given by Troelstra and van Dalen [1988a].

First-order models A *first-order model with equality* is a tuple $(D, =, R, F)$. The purpose of a first-order model is to interpret the first-order fragment of the logic, specifically first-order quantification, functions, and relations. D is a set, the *domain* of individuals. Semantically, quantification in the logic ranges over these individuals. R is a set $\{r_i \mid i \in I\}$ of relations on D, indexed by set I. Likewise, F is a set $\{f_j \mid j \in J\}$ of functions on D, indexed by set J. The sets I and J here stand for sets of *relation symbols* and *function symbols*, respectively, that are used in the logic. That is, this is the set of first-order functions and first order relations that f and r can range over in Figure 3.1, respectively. There is a distinguished equality relation $=$, which is an equivalence relation on D, such that equal individuals are indistinguishable by relations and functions.

To interpret first-order variables, the semantics employs *valuation* functions, which map variables to individuals. We write $v(x)$ to denote the individual that variable x represents in valuation v.

Constructive models A *constructive model* is a tuple $(W, \leq, s)$. The purpose of constructive models is to extend first-order models to interpret the constructive fragment of the logic, specifically implication and universal quantification. W is a set of *possible worlds*. We denote an individual world as w. Intuitively, a world w represents the *state of knowledge* of a constructive reasoner. The *constructive accessibility relation* $\leq$ is a partial order on W. If $w \leq w'$, then the constructive reasoner's state of knowledge could grow from w to w'. But unlike in classical logic, the reasoner need not commit to a formula φ being either true or false at a world. Suppose that at world w', where $w \leq w'$, the reasoner concludes that φ

holds. And at world w'', where $w \leq w''$, the reasoner concludes that $\neg\varphi$ holds. But at world w, the reasoner has not yet concluded that either φ or $\neg\varphi$ holds. Then Excluded Middle ($\varphi \vee \neg\varphi$) doesn't hold at w.

Function s is the *first-order interpretation function*. It assigns a first-order model $(D_w, =_w, R_w, F_w)$ to each world w. Let the individual elements of R_w be denoted $\{r_{i,w} \mid i \in I\}$; likewise for F_w, as $\{f_{j,w} \mid j \in J\}$. Thus, s enables a potentially different first-order interpretation at each world. But to help ensure that the constructive reasoner's state of knowledge only grows—hence never invalidates a previously admitted construction—we require s to be monotonic with respect to $\leq$. That is, if $w \leq w'$ then (i) $D_w \subseteq D_{w'}$, (ii) $d =_w d'$ implies $d =_{w'} d'$, (iii) $r_{i,w} \subseteq r_{i,w'}$, and (iv) for all tuples $\vec{d}$ of individuals in D_w, it holds that $f_{j,w}(\vec{d}) =_w f_{j,w'}(\vec{d})$.

It's natural to wonder why we chose to introduce possible worlds into the semantics here after arguing against them at the beginning of this chapter. Note, though, that the worlds in the constructive model are being used to model only the constructive reasoner—which we might think of as the guard, who exists outside of the logic and attempts to ascertain the truth of formulae—not any of the principals reasoned about inside the logic. Moreover, we have not introduced any accessibility relations for principals, but only a single accessibility relation for the constructive reasoner. So the arguments from the beginning of this chapter don't apply. It would be possible to eliminate our usage of possible worlds by employing a *Heyting algebra semantics* [Troelstra and van Dalen, 1988b] of constructive logic. But possible worlds blend better with our eventual introduction of accessibility relations for principals in Section 3.2.

It's also natural to wonder why FOCAL is constructive rather than classical. Schneider et al. [2011] write that constructivism preserves evidence: "Constructive

logics are well suited for reasoning about authorization... because constructive proofs include all of the evidence used for reaching a conclusion and, therefore, information about accountability is not lost. Classical logics allow proofs that omit evidence." They argue that Excluded Middle, used as an axiom in a proof, would omit evidence by failing to indicate whether access was granted on the basis of φ holding or $\neg\varphi$ holding. Garg and Pfenning [2006] also champion the notion of evidence in authorization logics, writing that "[constructive logics] keep evidence contained in proofs as direct as possible." So we chose to make FOCAL constructive for the sake of evidence. Regardless, we believe that a classical version of FOCAL could be created without difficulty.

Belief models A *belief model* is a tuple $(W, \leq, s, P, \omega)$. The purpose of belief models is to extend constructive models to interpret says and speaksfor. The first part of a belief model, $(W, \leq, s)$, must be a constructive model. The next part, P, is the set of principals. Although individuals can vary from world to world in a model, the set of principals is fixed across the entire model. Assuming a fixed set of principals is consistent with other authorization logics [Garg, 2008; Garg and Abadi, 2008; Genovese et al., 2012], with constructive multimodal logics [Simpson, 1994; Wijesekera, 1990] (which have a fixed set of modalities), and with classical multimodal epistemic logics [Fagin et al., 1995] (which have an indexed set of modalities, typically denoted K_i, where the index set is fixed)—even though constructivist philosophy might deem it more sensible to allow P to grow with $\leq$.

Because we make no syntactic distinction between individuals and principals, all principals must also be individuals: P must be a subset of D_w for every w. First-order quantification can therefore range over individuals as well as principals. For example, to quantify over all principals, we can write $(\forall\, x \;:\; \mathit{IsPrin}(x) \Rightarrow \varphi)$,

where *IsPrin* is a relation that holds for all $x \in P$. Nonetheless, this does not constitute truly intuitionistic quantification, because the domain of principals is constant. Quantification over a non-constant domain of principals is theoretically of interest, but we know of no authorization logic that has used it.

We define an equality relation $\doteq$ on principals, such that principals are equal if and only if they are equal at all worlds. Formally, $p \doteq p'$ if and only if, for all w, it holds that $p =_w p'$.

The final part of a belief model, a worldview function ω, yields the beliefs of a principal p: the set of formulas that p believes to hold in world w under first-order valuation v is $\omega(w, p, v)$. For the sake of simplicity, the beginning of this chapter used notation $\omega(p)$ when first presenting the idea of worldviews. Now that we're being precise, we also include w and v as arguments. To ensure that the constructive reasoner's knowledge grows monotonically, worldviews must be monotonic with respect to $\leq$:

Belief Frame Condition 1 (Worldview Monotonicity)**.** If $w \leq w'$ then $\omega(w, p, v) \subseteq \omega(w', p, v)$.

To ensure that whenever principals are equal they have the same worldview, we require the following:

Belief Frame Condition 2 (Worldview Equality)**.** We say that p and p' are *worldview equal*, and write $p \doteq p'$, if for all w and v, $\omega(w, p, v) = \omega(w, p', v)$.

We also require the following conditions to ensure that valuations cannot cause worldviews to distinguish alpha-equivalent formulas:

$B, w, v \models \mathsf{true}$		always
$B, w, v \models \mathsf{false}$		never
$B, w, v \models r_i(\tau_1, \ldots, \tau_n)$	iff	$(\mu(\tau_1), \ldots, \mu(\tau_n)) \in r_{i,w}$
$B, w, v \models \tau_1 = \tau_2$	iff	$\mu(\tau_1) =_w \mu(\tau_2)$
$B, w, v \models \varphi_1 \wedge \varphi_2$	iff	$B, w, v \models \varphi_1$ and $B, w, v \models \varphi_2$
$B, w, v \models \varphi_1 \vee \varphi_2$	iff	$B, w, v \models \varphi_1$ or $B, w, v \models \varphi_2$
$B, w, v \models \varphi_1 \Rightarrow \varphi_2$	iff	for all $w' \geq w : B, w', v \models \varphi_1$ implies $B, w', v \models \varphi_2$
$B, w, v \models \neg\varphi$	iff	for all $w' \geq w : B, w', v \not\models \varphi$
$B, w, v \models (\forall\, x : \varphi)$	iff	for all $w' \geq w, d \in D_{w'} : B, w', v[x \mapsto d] \models \varphi$
$B, w, v \models (\exists\, x : \varphi)$	iff	there exists $d \in D_w : B, w, v[x \mapsto d] \models \varphi$
$B, w, v \models \tau\ \mathsf{says}\ \varphi$	iff	$\varphi \in \omega(w, \mu(\tau), v)$
$B, w, v \models \tau_1\ \mathsf{speaksfor}\ \tau_2$	iff	for all $w' \geq w : \omega(w', \mu(\tau_1), v) \subseteq \omega(w', \mu(\tau_2), v)$

Figure 3.2: FOCAL validity judgment for belief semantics

Belief Frame Condition 3 (Worldview Valuations)**.**

1. Assume $x \notin FV(\varphi)$. Then $\varphi \in \omega(w, p, v)$ if and only if $\varphi \in \omega(w, p, v[x \mapsto d])$ for all $d \in D_w$.

2. Assume $x \in FV(\varphi)$ and $y \notin FV(\varphi)$. Then, $\varphi \in \omega(w, p, v[x \mapsto d])$ if and only if $\varphi[x \mapsto y] \in \omega(w, p, v[y \mapsto d])$ for all $d \in D_w$, where $\varphi[x \mapsto y]$ denotes the capture-avoiding substitution of x with y in formula φ.

Condition (1) ensures that if x is irrelevant to φ, then the value of x is also irrelevant to whether p believes φ. Condition (2) ensures that if x is relevant to φ, then only its value—not its name—is relevant to whether p believes φ.

These three frame conditions are almost enough to define well-formed belief models. We need just one more condition. However, we must define semantic validity in order to express that condition.

3.1.2 Semantic validity

Figure 3.2 gives a belief semantics of FOCAL. The validity judgment is written $B, w, v \models \varphi$ where B is a belief model and w is a world in that model. As is standard, $B, v \models \varphi$ holds if and only if, for all w, it holds that $B, w, v \models \varphi$; whenever $B, v \models \varphi$, then φ is a *valuation-necessary* formula. Likewise, $B \models \varphi$ holds if and only if, for all w and v, it holds that $B, w, v \models \varphi$; whenever $B \models \varphi$, then φ is a *necessary* formula in model B. $\models \varphi$ holds if and only if, for all B, it holds that $B \models \varphi$; and whenever $\models \varphi$, then φ is a *validity*. We use $B, w, v \models \Gamma$, where Γ is a set of formulas, to denote that for all $\psi \in \Gamma$, it holds that $B, w, v \models \psi$. Finally, $\Gamma \models \varphi$ holds if and only if, for all B, w, and v, it holds that $B, w, v \models \Gamma$ implies $B, w, v \models \varphi$; whenever $\Gamma \models \varphi$, then φ is a *logical consequence* of Γ.

The semantics relies on an auxiliary *interpretation* function μ that maps syntactic terms τ to semantic individuals:

$$\mu(x) = v(x)$$
$$\mu(f_j(\tau_1, \ldots, \tau_n)) = f_{j,w}(\mu(\tau_1), \ldots, \mu(\tau_n))$$

Implicitly, μ is parameterized on belief model B, world w, and valuation v, but for notational simplicity we omit writing these as arguments to μ unless necessary for disambiguation. Variables x are interpreted by looking up their value in v; functions f_j are interpreted by applying their first-order interpretation $f_{j,w}$ at world w to the interpretation of their arguments.

The first-order, constructive fragment of the semantics is routine. The semantics of says is the intuitive semantics we wished for at the beginning of this chapter. A principal $\mu(\tau)$ says φ exactly when φ is in that principal's worldview $\omega(w, \mu(\tau), v)$. And a principal $\mu(\tau_1)$ speaks for another principal $\mu(\tau_2)$ exactly

when, in all constructively accessible worlds, everything $\mu(\tau_1)$ says $\mu(\tau_2)$ also says.

Note that some syntactic terms may represent individuals that are not principals. For example, the integer 42 is presumably not a principal in P, but it could be an individual in some domain D_w. An alternative would be to make FOCAL a two-sorted logic, with one sort for individuals and another sort for principals. Instead, we allow individuals who aren't principals to have beliefs, because it simplifies the definition of the logic. The worldviews of non-principal individuals contain all formulas. Formally, for any individual d such that $d \notin P$, and for any world w, valuation v, and formula φ, it holds that $\varphi \in \omega(w, d, v)$.

We impose a few *well-formedness* conditions on worldviews in this semantics, in addition to Worldview Monotonicity and Worldview Equality. Worldviews must be *closed under logical consequence*—that is, principals must believe all the formulas that are a consequence of their beliefs.

Belief Frame Condition 4 (Worldview Closure)**.** If $\Gamma \subseteq \omega(w, p, v)$ and $\Gamma \models \varphi$, then $\varphi \in \omega(w, p, v)$.

Worldview Closure means that principals are *fully logically omniscient* [Fagin et al., 1995]. With its known benefits and flaws [Parikh, 1987; Stalnaker, 1991], this has been a standard assumption in authorization logics since their inception [Lampson et al., 1991].

The remaining well-formedness conditions are optional, in the sense that they are necessary only to achieve soundness of particular proof rules in Section 3.4. Eliminate those rules, and the following conditions would be eliminated.

Worldviews must ensure that says is a *transparent* modality. That is, for any principal p, it holds that $p \mathsf{\ says\ } \varphi$ exactly when $p \mathsf{\ says\ } (p \mathsf{\ says\ } \varphi)$:

Belief Frame Condition 5 (Says Transparency)**.** $\varphi \in \omega(w, \mu(\tau), v)$ if and only if τ says $\varphi \in \omega(w, \mu(\tau), v)$.

So says supports *positive introspection*: if p believes that φ holds, then p is aware of that belief, therefore p believes that p believes that φ holds. The converse of that holds as well. Recent authorization logics include transparency [Abadi, 2008; Schneider et al., 2011], and it is well known (though sometimes vigorously debated) in epistemic logic [Hintikka, 1962; Hughes and Cresswell, 1996]. Says Transparency corresponds to rules SAYS-LI and SAYS-RI in Figure 3.5.

Worldviews must enable principals to delegate, or *hand-off*, to other principals: if a principal q believes that q speaksfor p, it should hold that p does speak for q. Hand-off, as the following axiom, existed in the earliest authorization logic [Lampson et al., 1991]:

$$(\tau' \text{ says } (\tau \text{ speaksfor } \tau')) \Rightarrow (\tau \text{ speaksfor } \tau') \tag{3.1}$$

To support it, we adopt a condition that ensures whenever q believes p speaks for q, then it really does:

Belief Frame Condition 6 (Belief Hand-off)**.** If $(p \text{ speaksfor } q) \in \omega(w, q, v)$ then $\omega(w, p, v) \subseteq \omega(w, q, v)$.

Belief Hand-off corresponds to rule SF-I in Figure 3.5.

3.2 Kripke Semantics

The Kripke semantics of FOCAL combines first-order constructive models with modal (Kripke) models [Fagin et al., 1995; Hughes and Cresswell, 1996; Simpson,

1994]. Similar semantic models have been explored before (see, for example, Wijesekera [1990], Genovese et al. [2012], and Garg [2008]). Indeed, the only non-standard part of our semantics is the treatment of speaksfor, and that part turns out to be a generalization of previous classical semantics.

Nonetheless, we are not aware of any authorization logic semantics that is equivalent to or subsumes our semantics. First-order and constructive models were already presented in Section 3.1, so we begin here with modal models.

3.2.1 Modal models

A *modal model* is a tuple $(W, \leq, s, P, A)$. The purpose of a modal model is to extend constructive models to interpret says and speaksfor. The first part of a modal model, $(W, \leq, s)$, must itself be a constructive model. The next part, P, is the set of *principals*. As with belief models, all principals must be individuals, so P must be a subset of D_w for every w. Principal equality relation $\doteq$ is defined just as in belief models. The final part of a modal model, A, is a set $\{\leq_p \mid p \in P\}$ of binary relations on W, called the *principal accessibility relations*.[4] If $w \leq_p w'$, then at world w, principal p considers world w' possible. To ensure that equal principals have the same beliefs, we require

Kripke Frame Condition 1 (Accessibility Equality)**.** If $p \doteq p'$, then $\leq_p = \leq_{p'}$.

As with the constructive accessibility relation $\leq$, we require s to be monotonic with respect to each principal accessibility relation $\leq_p$. This requirement enforces a kind of constructivity on each principal p, such that from a world in which

[4]In our notation, an unsubscripted $\leq$ always denotes the constructive relation, and a subscripted $\leq$ always denotes a principal relation.

$$
\begin{array}{lll}
K, w, v \models \tau \text{ says } \varphi & \text{iff} & \text{for all } w', w'' : w \leq w' \leq_{\mu(w',\tau)} w'' \\
 & & \text{implies } K, w'', v \models \varphi \\
K, w, v \models \tau_1 \text{ speaksfor } \tau_2 & \text{iff} & \leq^w_{\mu(\tau_1)} \supseteq \leq^w_{\mu(\tau_2)} \\
K, w, v \models \ldots & \text{iff} & \textit{same as Figure 3.2, but substituting } K \textit{ for } B
\end{array}
$$

Figure 3.3: FOCAL validity judgment for Kripke semantics

individual d is constructed, p cannot consider possible any world in which d has not been constructed. Unlike $\leq$, none of the $\leq_p$ are required to be partial orders: they are not required to satisfy reflexivity, anti-symmetry, or transitivity.

In epistemic logics, the properties of what we call the "principal accessibility relations" determine what kind of knowledge is modeled [Fagin et al., 1995]. If, for example, these relations must be reflexive, then the logic models *veridical* knowledge: if p says φ, then φ indeed holds. But that is not the kind of knowledge we seek to model with FOCAL, because principals may say things that in fact do not hold. So what are the right properties, or *frame conditions*, to require of our principal accessibility relations? We briefly delay presenting them, so that we can present the Kripke semantics of FOCAL.

3.2.2 Semantic validity

Figure 3.3 gives a Kripke semantics of FOCAL. The validity judgment is written $K, w, v \models \varphi$ where K is a modal model and w is a world in that model. Only the judgments for the says and speaksfor connectives are given in Figure 3.3. For the remaining connectives, the Kripke semantics is the same as the belief semantics in Figure 3.2. Interpretation function μ remains unchanged from Section 3.1, except that it is now implicitly parameterized by K instead of B.

To understand the semantics of says, first observe the following. Suppose that, for all worlds w', it holds that $w \leq w'$ implies $w = w'$.[5] Then the semantics of says simplifies to

$$K, w, v \models \tau \text{ says } \varphi \quad \text{if and only if} \quad \text{for all } w'' : w \leq_{\mu(\tau)} w'' \text{ implies } K, w, v \models \varphi,$$

which is the standard semantics of $\Box$ in classical modal logic [Hughes and Cresswell, 1996]: a principal believes a formula holds whenever that formula holds in all accessible worlds. The purpose of the quantification over w', where $w \leq w'$, in the unsimplified semantics of says is to achieve *monotonicity* of the constructive reasoner:

Lemma 2. *If $K, w, v \models \varphi$ and $w \leq w'$ then $K, w', v \models \varphi$.*

Proof. By structural induction on φ. This proof has been mechanized in Coq [Hirsch and Clarkson, 2013b]. □

That is, whenever φ holds at a world w, if the constructive reasoner is able to reach an extended state of knowledge at world w', then φ should continue to hold at w'. Without the quantification over w' in the semantics of says, monotonicity is not guaranteed to hold. Constructive modal logics have, unsurprisingly, also used this semantics for $\Box$ Simpson [1994]; Wijesekera [1990], and a similar semantics has been used in authorization logic Garg [2008].

Note that, if there do not exist any worlds w' and w'' such that $w \leq w' \leq_{\mu(\tau)} w''$, then at w, principal τ will say any formula φ, including false. When a principal says false at world w, we deem that principal *compromised* at w.

[5] This condition corresponds to the axiom of excluded middle, hence its imposition creates a classical variant of FOCAL. So it makes sense that adding the frame condition would result in the classical semantics of $\Box$.

As for the semantics of speaksfor, it might be tempting to try defining it as syntactic sugar:

$$\tau_1 \text{ speaksfor } \tau_2 \quad \equiv \quad \forall \varphi : \tau_1 \text{ says } \varphi \Rightarrow \tau_2 \text{ says } \varphi$$

However, the formula on the right-hand side is not a well-formed formula of FOCAL, because it quantifies over formulae.[6]

Instead, the FOCAL semantics of speaksfor generalizes the classical Kripke semantics of speaksfor [Abadi et al., 1993; Howell, 2000]. Classically,

$$K, w, v \models \tau_1 \text{ speaksfor } \tau_2 \quad \text{if and only if} \quad \leq_{\mu(\tau_1)} \supseteq \leq_{\mu(\tau_2)}. \tag{3.2}$$

That is, the accessibility relation of τ_1 must be a superset of the accessibility relation of τ_2. However, that definition does not account for constructive accessibility, and it even turns out to interact badly with hand-off (see Section 3.2.4). We therefore relax the classical semantics of speaksfor:

$$K, w, v \models \tau_1 \text{ speaksfor } \tau_2 \quad \text{if and only if} \quad \leq^w_{\mu(\tau_1)} \supseteq \leq^w_{\mu(\tau_2)} \tag{3.3}$$

where $\leq^w_{\mu(p)}$ is defined to be $\leq_p |_{[w]_p}$,[7] and $[w]_p$ is defined to be the set of worlds w' such that w' is reachable from w, or vice-versa, by relation $(\leq \cup \leq_p)^*$. Note that whenever $[w]_p$ equals W (as it would in classical logic[8]), it holds that $\leq^w_{\mu(p)}$ equals $\leq_p$.

The validity judgment for FOCAL is therefore quite standard, except for the semantics of speaksfor, which generalizes classical logic. Although we would prefer to adopt a well-known constructive semantics of speaksfor, neither of the two

[6]It is possible [Garg and Abadi, 2008; Schneider et al., 2011] to instead use second-order quantifiers to achieve a direct interpretation. That solution would unnecessarily complicate our semantics by introducing second-order quantifiers just for the sake of defining speaksfor.

[7]If R is a binary relation on set A, then $R|_X$ is the *restriction* of R to A, where $X \subseteq A$. That is, $R|_X = \{(x, x') \mid (x, x') \in R \text{ and } x \in X \text{ and } x' \in X\}$.

[8]When frame condition $\leq \; = W \times W$ is imposed, constructive logic collapses to classical. Under that condition, every world w' would be reachable from w, hence $[w]_p = W$.

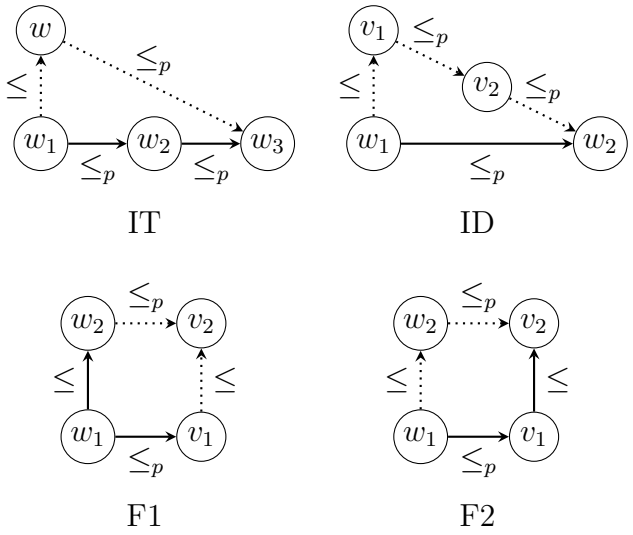


Figure 3.4: Frame conditions for Kripke semantics

such semantics we are aware of seems to work for FOCAL: ICL [Genovese et al., 2012] would impose an axiom called Unit that we do not want to include (see Section 3.4.1), and $\mathrm{BL}_{\mathrm{sf}}$ does not include hand-off (3.1), which we want to optionally support (see Sections 3.1.2 and 3.2.3).

3.2.3 Frame conditions

We now begin the discussion of frame conditions referenced in Section 3.2.1 of the frame conditions for FOCAL. The first two frame conditions we impose help to ensure Says Transparency.

Kripke Frame Condition 2 (IT)**.** If $w_1 \leq_p w_2 \leq_p w_3$, then there exists a w such that $w_1 \leq w \leq_p w_3$.

Kripke Frame Condition 3 (ID)**.** If $w_1 \leq_p w_2$, then there exists a v_1 and v_2 such that $w \leq v_1 \leq_p v_2 \leq_p w_2$.

Figure 3.4 depicts these conditions; dotted lines indicate existentially quantified

edges. IT helps to guarantee if p says φ then p says $(p \text{ says } \varphi)$; ID does the converse.[9]

Note how, if $w = w'$, the conditions reduce to the classical definitions of transitivity and density. Those classical conditions are exactly what guarantee transparency in classical modal logic.

IT and ID are not quite sufficient to yield transparency. But by also imposing the following frame condition, we do achieve transparency:[10]

Kripke Frame Condition 4 (F2)**.** If $w_1 \leq_p w_2 \leq w_3$, then there exists a w such that $w_1 \leq w \leq_p w_3$.

F2 is depicted in Figure 3.4. It is difficult to motivate F2 solely in terms of authorization logic, though it has been proposed in several Kripke semantics for constructive modal logics [Ewald, 1986; Fischer Servi, 1981; Plotkin and Stirling, 1986; Simpson, 1994]. But there are two reasons why F2 is desirable for FOCAL:

- Assuming F2 holds, IT and ID are not only sufficient but also necessary conditions for transparency—a result that follows from work by Plotkin and Stirling [1986]. So in the presence of F2, transparency in FOCAL is precisely characterized by IT and ID.
- Suppose FOCAL were to be extended with a $\Diamond$ modality. It could be written τ suspects φ, with semantics $K, w, v \models \tau$ suspects φ if and only if there exists w' such that $w \leq_{\mu(\tau)} w'$ and $K, w', v \models \varphi$. We would want says and suspects to interact smoothly. For example, it would be reasonable to expect

[9] IT and ID are abbreviations for intuitionistic transitivity and intuitionistic density. We use the term "intuitionistic" instead of "constructive" just to avoid confusion: CT might be read as classical or constructive transitivity.

[10] F2 is the name given this condition by Simpson [1994].

that $\neg(\tau$ suspects $\varphi)$ implies τ says $\neg\varphi$. For if τ does not suspect φ holds anywhere, then τ should believe $\neg\varphi$ holds. Condition F2 guarantees that implication [Plotkin and Stirling, 1986]. So F2 prepares FOCAL for future extension with a suspects modality.

Were suspects to be added to FOCAL, it would also be desirable to impose a fourth frame condition: if $w \leq w'$ and $w \leq_p v$, then there exists a v' such that $v \leq v'$ and $w' \leq_p v'$. This condition, named F1 by Simpson [1994], guarantees [Plotkin and Stirling, 1986] that τ suspects φ implies $\neg(\tau$ says $\neg\varphi)$. It also guarantees monotonicity (see Lemma 2) for suspects. Figure 3.4 depicts F1. Simpson [1994, p. 51] argues that F1 and F2 could be seen as fundamental, not artificial, frame conditions for constructive modal logics.

To ensure the validity of hand-off, we impose the following frame condition:

Kripke Frame Condition 5 (H)**.** For all principals p and worlds w, if there do not exist any worlds w' and w'' such that $w \leq w' \leq_p w''$, then, for all p', it must hold that $\leq^w_{\mu(p)} \subseteq \leq^w_{\mu(p')}$.

This condition guarantees that if a principal p becomes compromised at world w, then the reachable component of its accessibility relation will be a subset of all other principals'. By the FOCAL semantics of speaksfor, all other principals therefore speak for p at w.

Each frame condition above was imposed, not for ad hoc purposes, but because of a specific need in the proof of the soundness result of Section 3.4. So with appropriate deletion of rules from the proof system, each of the above frame conditions could be eliminated. IT and ID should be removed if rules SAYS-LI and SAYS-RI

(from Figure 3.5) are removed; F2 should be removed if rule SAYS-LRI is removed; and H should be removed if rule SF-I is removed.

Finally, we impose one additional condition to achieve the equivalence results (Theorem 8 and Lemma 9) of Section 3.3:

Kripke Frame Condition 6 (WSF)**.** $K, w, v \models \tau_1$ speaksfor τ_2 if and only if, for all φ, if $K, w, v \models \tau_1$ says φ then $K, w, v \models \tau_2$ says φ.

This condition restricts the class of Kripke models to those where speaksfor is the *weak speaksfor* connective Abadi et al. [1993]; Howell [2000]. Note that the semantics of speaksfor ensures one direction already: if $K, w, v \models \tau_1$ speaksfor τ_2, then for all φ, if $K, w, v \models \tau_1$ says φ, then $K, w, v \models \tau_2$ says φ. Thus, this condition only restricts the Kripke models to those where, if the beliefs of τ_2 subsume those of τ_1, then τ_1 trusts τ_2. In some sense, this is a strong assumption, since one can imagine trust being a stronger relationship then "merely" saying the same things. However, our belief semantics exhibits this property, and so we need to ensure that the Kripke semantics does as well in order to achieve our equivalence results.

3.2.4 Defining Speaksfor

Abadi [2008] presents several strange consequences of classical authorization logic. Here is yet another that results when we try to use the classical definition of speaksfor (3.2) in a constructive setting:

Example 2. Consider a world w. Suppose there do not exist any worlds w' and w'' such that $w \leq w' \leq_{\mu(\tau)} w''$. Then at world w, principal τ is compromised: it says false, and also says any other formula φ. Then, for any principal τ', it holds

that $K, w, v \models \tau$ says $(\tau'$ speaksfor $\tau)$. By hand-off, $K, w, v \models \tau'$ speaksfor τ. By the classical semantics of speaksfor, we have $\leq_{\mu(\tau')} \supseteq \leq_{\mu(\tau)}$. So τ's accessibility relation must be a subset of all other principal's accessibility relations. In the extreme case, if there is a principal whose accessibility relation is empty, τ's relation must also be empty.

Therefore, if there ever is any world w at which principal τ is compromised, then τ's accessibility relation must be empty. That means if τ is compromised at one world, τ must be compromised at all worlds.

As a result, the constructive reasoner is immediately forced to recognize that a principal is compromised, even if the reasoner is in a minimal state of knowledge (i.e., at a world w at which there do not exist any worlds v such that $v \leq w$.) The reasoner is not allowed to wait until some greater state of knowledge to discover that a principal is compromised. This seems intuitionistically undesirable.

But with FOCAL's definition of speaksfor (Section 3.3), only the components of the accessibility relations that are locally reachable from w need to be considered. So a principal could be entirely compromised in some set of worlds not reachable from w, but that principal need not be compromised at w.

3.3 Semantic Transformation

We have now given two classes of models for FOCAL, belief models (Section 3.1) and Kripke models (Section 3.2). Naturally, the question arises: how are these models related? It turns out that Kripke models can be soundly transformed into belief structures, as we now explain.

Given a modal model K, there is a natural way to construct a belief model from it: assign each principal a worldview containing exactly the formulae that the principal says in K. Call this construction *k2b*, and let $k2b(K)$ denote the resulting belief model.

To give a precise definition of *k2b*, we need to introduce a new notation. Given semantic principal $p \in P$, formula p says φ is not necessarily well-formed, because p is not necessarily a syntactic term. So let $K, w, v \models \hat{p}$ says φ be defined as follows: for all w' and w'' such that $w \leq w' \leq_p w''$, it holds that $K, w'', v \models \varphi$. This definition simply unrolls the semantics of says to produce something well-formed.[11]

The precise definition of *k2b* is as follows: if $K = (W, \leq, s, P, A)$, then $k2b(K)$ is belief model $(W, \leq, s, P, \omega)$, where $\omega(w, p, v)$ is defined to be $\{\varphi \mid K, w, v \models \hat{p}$ says $\varphi\}$.

Our first concern is whether $k2b(K)$ produces a belief model that is equivalent to K. In particular, a formula should be valid in K if and only if it is valid in $k2b(K)$. Construction *k2b* does produce equivalent models:

Theorem 8. *For all K, w, v, and φ, $K, w, v \models \varphi$ if and only if $k2b(K), w, v \models \varphi$.*

Proof. $K, w, v \models \varphi$ implies $k2b(K), w, v \models \varphi$. All of the cases except says and speaksfor are straightforward, because those are the only two cases where the interpretation of formulae differs in the two semantics.

- Case $\varphi = \tau$ says ψ. Suppose $K, w, v \models \tau$ says ψ. By the definition of *k2b*, formula $\psi \in \omega(w, \mu(\tau), v)$. By the belief semantics of says, it must hold that $k2b(K), w, v \models \tau$ says ψ.

[11] Another solution would be to stipulate that every principal p can be named by a term $\hat{p}$ in the syntax.

- Case $\varphi = \tau$ speaksfor τ'. Assume $K, w, v \models \tau$ speaksfor τ'. We need to show that, for all $w' \geq w$, it holds that $\omega(w', \mu(\tau), v) \subseteq \omega(w', \mu(\tau'), v)$. So let w' and ψ be arbitrary such that $w' \geq w$ and $\psi \in \omega(w', \mu(\tau), v)$, and we'll show that $\psi \in \omega(w', \mu(\tau'), v)$. By the definition of *k2b*, it holds that $K, w', v \models \tau$ says ψ. Note that, by Lemma 2 and our original assumption, we have that $K, w', v \models \tau$ speaksfor τ'. From those last two facts, and from the Kripke semantics of says and speaksfor, it follows that $K, w', v \models \tau'$ says ψ. By the definition of *k2b*, it therefore holds that $\psi \in \omega(w', \mu(\tau'), v)$.

Second, we show the backward direction: if $k2b(K), w, v \models \varphi$, then $K, w, v \models \varphi$. Again, all of the cases except says and speaksfor are straightforward, because those are the only two cases where the interpretation of formulae differs in the two semantics.

- Case $\varphi = \tau$ says ψ. Suppose $k2b(K), w, v \models \tau$ says ψ. By the belief semantics of says, we have that $\psi \in \omega(w, \mu(\tau), v)$. By the definition of *k2b*, it holds that $K, w, v \models \tau$ says ψ.
- Case $\varphi = \tau$ speaksfor τ'. Assume $k2b(K), w, v \models \tau$ speaksfor τ'. By the belief semantics of speaksfor, we have that, for all $w' \geq w$, it holds that $\omega(w', \mu(\tau), v) \subseteq \omega(w', \mu(\tau'), v)$. By setting w' as w, we know that $\omega(w, \mu(\tau), v) \subseteq \omega(w, \mu(\tau'), v)$. By the definitions of *k2b* and subset, it follows that, for all φ, if $K, w, v \models \tau$ says φ then $K, w, v \models \tau'$ says φ. By WSF, we therefore have that $K, w, v \models \tau$ speaksfor τ'. □

Our second concern is whether $k2b(K)$ satisfies all the conditions required by Section 3.1: Worldview Monotonicity, Worldview Equality, Worldview Closure, Says Transparency, and Belief Hand-off. If a belief model B does satisfy these

conditions, then B is *well-formed*. And modal model K is well-formed if it satisfies all the conditions required by Section 3.2: Accessibility Equality, IT, ID, F2, H, and WSF. Construction *k2b* does, indeed, produce well-formed belief models from well-formed Kripke models:

Theorem 9. *For any well-formed Kripke model K, k2b(K) is a well-formed Belief model.*

Proof. Let $B = k2b(K)$. For B to be well-formed it must satisfy several conditions, which were defined in Section 3.1. We now show that these hold for any such B constructed by *k2b*.

1. Worldview Monotonicity. Assume $w \leq w'$ and $\varphi \in \omega(w, p, v)$. By the latter assumption and the definition of *k2b*, we have that $K, w, v \models \hat{p}$ says φ. From Lemma 2, it follows that $K, w', v \models \hat{p}$ says φ. By the definition of *k2b*, it then holds that $\varphi \in \omega(w', p, v)$. Therefore $\omega(w, p, v) \subseteq \omega(w', p, v)$.

2. Worldview Equality. Assume $p \doteq p'$. Then by Accessibility Equality, $\leq_p$ equals $\leq_{p'}$. By the Kripke semantics of says, it follows that $K, w, v \models p$ says φ if and only if $K, w, v \models p'$ says φ. By the definition of *k2b*, therefore, $\omega(w, p, v) = \omega(w, p', v)$.

3. Worldview Closure. Assume $\Gamma \subseteq \omega(w, p, v)$ and $\Gamma \models \varphi$, that is, φ is a logical consequence of Γ in belief structure B. By the definition of *k2b*, we have $\omega(w, p, v) = \{\varphi \mid K, w, v \models \hat{p}$ says $\varphi\}$. So for all $\psi \in \Gamma$, it holds that $K, w, v \models \hat{p}$ says ψ. By the Kripke semantics of says, it follows that for all w' and w'' such that $w \leq w' \leq_p w''$, it holds that $K, w'', v \models \psi$. Thus $K, w'', v \models \Gamma$. So $k2b(K), w'', v \models \Gamma$ by Theorem 8. By our initial assumption that $\Gamma \models \varphi$, it follows that $k2b(K), w'', v \models \varphi$. Again applying

Theorem 8, we have that $K, w'', v \models \varphi$. By the Kripke semantics of says, it follows that $K, w, v \models \hat{p}$ says φ. Therefore, by the definition of *k2b*, we have $\varphi \in \omega(w, p, v)$.

4. Says Transparency. We prove the "if and only if" by proving both directions independently.

   ($\Rightarrow$) Assume $\varphi \in \omega(w, p, v)$. By the definition of *k2b*, $K, w, v \models \hat{p}$ says φ. From IT and F2, it follows that $K, w, v \models \hat{p}$ says ($\hat{p}$ says φ). By the definition of *k2b*, therefore, ($\hat{p}$ says φ) $\in \omega(w, p, v)$.

   ($\Leftarrow$) Assume ($\hat{p}$ says φ) $\in \omega(w, p, v)$. By the definition of *k2b*, it holds that $K, w, v \models \hat{p}$ says ($\hat{p}$ says φ). From ID, it follows that $K, w, v \models \hat{p}$ says φ. By the definition of *k2b*, therefore, $\varphi \in \omega(w, p, v)$.

5. Belief Hand-off. We actually prove a stronger result—an "if and only if" rather than just an "if". By the definitions of subset and *k2b*, we have that $\omega(w, p, v) \subseteq \omega(w, q, v)$ holds if and only if for all φ, if $K, w, v \models \hat{p}$ says φ then $K, w, v \models \hat{q}$ says φ. By WSF, $K, w, v \models \hat{p}$ speaksfor $\hat{q}$. By Lemma 3 (proven below), that is equivalent to $K, w, v \models \hat{q}$ says ($\hat{p}$ speaksfor $\hat{q}$). By the definition of *k2b*, that holds if and only if $\hat{q}$ speaksfor $\hat{p} \in \omega(w, q, v)$.

□

Lemma 3. *In the Kripke semantics,* $\models \hat{q}$ says ($\hat{p}$ speaksfor $\hat{q}$) $\iff$ $\hat{p}$ speaksfor $\hat{q}$.

Proof. The proof of this fact has been mechanized in Coq [Hirsch and Clarkson, 2013b]. □

We might wonder whether there is a construction that can soundly transform belief models into Kripke models. Consider trying to transform the following belief

model B into a Kripke model:

> B has a single world w and a proposition (i.e., a nullary relation) X, such that, for all v, it holds that $B, w, v \not\models X$. Suppose that principal p's worldview contains X—i.e., for all v, it holds that $X \in \omega(w, p, v)$—and that p's worldview does not contain false. By the semantics of says, it holds that $B, w, v \models p \mathsf{\ says\ } X$.

When transforming B to a Kripke model K, what edges could we put in $\leq_p$? There are only two choices: $\leq_p$ could be empty, or $\leq_p$ could contain the single edge (w, w). If $\leq_p$ is empty, then p is compromised, hence p says false. That contradicts our assumption that false is not in p's worldview. If $w \leq_p w$, then for w' and w'' such that $w \leq w' \leq_p w''$, it does not hold that $K, w'', v \models X$, because w and w'' can only be instantiated as w, and because $B, w, v \not\models X$. Hence p does not say X. That contradicts our assumption that X is in p's worldview. So we cannot construct an accessibility relation $\leq_p$ that causes the resulting Kripke semantics to preserve validity of formulae from the belief semantics.

There is, therefore, no construction that can soundly transform belief models into Kripke models—unless, perhaps, the set of worlds is permitted to change. We conjecture that it is possible to synthesize a new set of possible worlds, and equivalence relations on them, yielding a Kripke model that preserves validity of formulas from the belief model.

3.4 Proof System

FOCAL's derivability judgment is written $\Gamma \vdash \varphi$ where Γ is a set of formulae called the *context*.[12] As is standard, we write $\vdash \varphi$ when Γ is the empty set. In that case, φ is a *theorem*. We write Γ, φ to denote $\Gamma \cup \{\varphi\}$.

Figure 3.5 presents the proof system. In it, $\varphi[x \mapsto \tau]$ denotes capture-avoiding substitution of τ for x in φ. The first-order fragment of the proof system is routine (e.g., Sørensen and Urzyczyn [2006], van Dalen [2004], and Negri and von Plato [2001]).[13] SAYS-LRI, SAYS-LI, and SAYS-RI use notation τ says Γ, which means that τ says all the formulae in set Γ. Formally, τ says Γ is defined as $\{\tau$ says $\varphi \mid \varphi \in \Gamma\}$.

The usual sequent calculus structural rules of contraction and exchange are admissible. But weakening (our rule WEAK) is not admissible: it must be directly included in the proof system, because the conclusions of SAYS-{LRI,LI,RI} capture their entire context Γ inside says.

SAYS-LRI corresponds Hughes and Cresswell [1996] to standard axiom K along with rule N from epistemic logic; SAYS-RI, to standard axiom 4; and SAYS-LI, to

[12] These formulas are *localized hypotheses*, which the proof system uses instead of the hypothetical judgments found in natural deduction systems. Similar to the left-hand side Γ of a sequent $\Gamma \Longrightarrow \Delta$, the localized hypotheses are assumptions being used to derive right-hand side Δ. However, we present our proof system in a natural-deduction style, with introduction and elimination rules. This hybrid system should be familiar, since it is the standard presentation of a type system.

[13] Under the usual constructive definition of $\neg\varphi$ as $\varphi \Rightarrow \mathsf{false}$, rules NOT-I and NOT-E are admissible and could be eliminated from the proof system.

$$\textsc{hyp}\ \frac{}{\Gamma, \varphi \vdash \varphi} \qquad \textsc{weak}\ \frac{\Gamma \vdash \varphi}{\Gamma, \psi \vdash \varphi}$$

$$\textsc{true-i}\ \frac{}{\Gamma \vdash \mathsf{true}} \qquad \textsc{false-e}\ \frac{\Gamma \vdash \mathsf{false}}{\Gamma \vdash \varphi}$$

$$\textsc{and-i}\ \frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} \qquad \textsc{and-le}\ \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} \qquad \textsc{and-re}\ \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}$$

$$\textsc{or-li}\ \frac{\Gamma \vdash \varphi_1}{\Gamma \vdash \varphi_1 \vee \varphi_2} \qquad \textsc{or-ri}\ \frac{\Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \vee \varphi_2}$$

$$\textsc{or-e}\ \frac{\Gamma, \varphi_1 \vdash \psi \qquad \Gamma, \varphi_2 \vdash \psi \qquad \Gamma \vdash \varphi_1 \vee \varphi_2}{\Gamma \vdash \psi}$$

$$\textsc{imp-i}\ \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \Rightarrow \psi} \qquad \textsc{imp-e}\ \frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \varphi \Rightarrow \psi}{\Gamma \vdash \psi}$$

$$\textsc{not-i}\ \frac{\Gamma, \varphi \vdash \mathsf{false}}{\Gamma \vdash \neg\varphi} \qquad \textsc{not-e}\ \frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \neg\varphi}{\Gamma \vdash \mathsf{false}}$$

$$\textsc{forall-i}\ \frac{\Gamma \vdash \varphi \qquad x \notin FV(\Gamma)}{\Gamma \vdash (\forall x : \varphi)} \qquad \textsc{forall-e}\ \frac{\Gamma \vdash (\forall x : \varphi)}{\Gamma \vdash \varphi[x \mapsto \tau]}$$

$$\textsc{exists-i}\ \frac{\Gamma \vdash \varphi[x \mapsto \tau]}{\Gamma \vdash (\exists x : \varphi)} \qquad \textsc{exists-e}\ \frac{\Gamma \vdash (\exists x : \varphi) \qquad \Gamma, \varphi \vdash \psi \qquad x \notin FV(\Gamma, \psi)}{\Gamma \vdash \psi}$$

$$\textsc{eq-r}\ \frac{}{\Gamma \vdash \tau = \tau} \qquad \textsc{eq-s}\ \frac{\Gamma \vdash \tau_1 = \tau_2}{\Gamma \vdash \tau_2 = \tau_1} \qquad \textsc{eq-t}\ \frac{\Gamma \vdash \tau_1 = \tau_2 \qquad \Gamma \vdash \tau_2 = \tau_3}{\Gamma \vdash \tau_1 = \tau_3}$$

$$\textsc{eq-fun}\ \frac{\Gamma \vdash \tau_i = \tau_i' \qquad i = 1, \ldots, n}{\Gamma \vdash f(\tau_1, \ldots, \tau_n) = f(\tau_1', \ldots, \tau_n')}$$

$$\textsc{eq-rel}\ \frac{\Gamma \vdash r(\tau_1, \ldots, \tau_n) \qquad \Gamma \vdash \tau_i = \tau_i' \qquad i = 1, \ldots, n}{\Gamma \vdash r(\tau_1', \ldots, \tau_n')}$$

$$\textsc{says-lri}\ \frac{\Gamma \vdash \varphi}{\tau\ \mathsf{says}\ \Gamma \vdash \tau\ \mathsf{says}\ \varphi} \qquad \textsc{says-li}\ \frac{\Gamma \vdash \tau\ \mathsf{says}\ \varphi}{\tau\ \mathsf{says}\ \Gamma \vdash \tau\ \mathsf{says}\ \varphi}$$

$$\textsc{says-ri}\ \frac{\tau\ \mathsf{says}\ \Gamma \vdash \varphi}{\tau\ \mathsf{says}\ \Gamma \vdash \tau\ \mathsf{says}\ \varphi}$$

$$\textsc{sf-i}\ \frac{\Gamma \vdash \tau_2\ \mathsf{says}\ (\tau_1\ \mathsf{speaksfor}\ \tau_2)}{\Gamma \vdash \tau_1\ \mathsf{speaksfor}\ \tau_2} \qquad \textsc{sf-e}\ \frac{\Gamma \vdash \tau_1\ \mathsf{speaksfor}\ \tau_2 \qquad \Gamma \vdash \tau_1\ \mathsf{says}\ \varphi}{\Gamma \vdash \tau_2\ \mathsf{says}\ \varphi}$$

$$\textsc{sf-r}\ \frac{}{\Gamma \vdash \tau\ \mathsf{speaksfor}\ \tau} \qquad \textsc{sf-t}\ \frac{\Gamma \vdash \tau_1\ \mathsf{speaksfor}\ \tau_2 \qquad \Gamma \vdash \tau_2\ \mathsf{speaksfor}\ \tau_3}{\Gamma \vdash \tau_1\ \mathsf{speaksfor}\ \tau_3}$$

Figure 3.5: FOCAL derivability judgment

the converse $C4$ Abadi [2008] of 4:

$$\begin{aligned} K: &\quad (p \text{ says } (\varphi \Rightarrow \psi)) \Rightarrow (p \text{ says } \varphi) \Rightarrow (p \text{ says } \psi) \\ N: &\quad \text{From } \varphi \text{ infer } p \text{ says } \varphi \\ 4: &\quad (p \text{ says } \varphi) \Rightarrow (p \text{ says } (p \text{ says } \varphi)) \\ C4: &\quad (p \text{ says } (p \text{ says } \varphi)) \Rightarrow (p \text{ says } \varphi) \end{aligned}$$

K and SAYS-LRI mean that *modus ponens* applies inside says. They correspond to Worldview Closure. $C4$ and 4, along with SAYS-LI and SAYS-RI, mean that p says $(p$ says $\varphi)$ is equivalent to p says φ; they correspond to Says Transparency in the belief semantics. In the Kripke semantics, SAYS-RI corresponds to IT; and SAYS-LI, to ID. (Note, we do not argue that 4 and $C4$ are necessary in authorization logics; we simply show how to support them.)

SF-I corresponds to hand-off (see Section 3.1). SF-E uses speaksfor to deduce beliefs. SF-R and SF-T state that speaksfor is reflexive and transitive.

3.4.1 Unit and Necessitation

There are two standard ways of "importing" beliefs into a principal's worldview. The first is rule N from Section 3.4, also known as the rule of Necessitation:

$$\frac{\vdash \varphi}{\vdash p \text{ says } \varphi}$$

The second is an axiom known as Unit:

$$\vdash \varphi \Rightarrow (p \text{ says } \varphi)$$

Though superficially similar, it is well-known that Necessitation and Unit lead to different theories. Abadi [2008] explores some of the proof-theoretic differences,

particularly some of the surprising consequences of Unit in classical authorization logic. In the example below, we focus on one difference that does not seem to have been explored in constructive authorization logic:

Example 3. Machines M_1 and M_2 execute processes P_1 and P_2, respectively. M_1 has a register R. Let Z be a proposition representing "register R is currently set to zero." According to Unit, $\vdash Z \Rightarrow (P_1 \text{ says } Z)$ and $\vdash Z \Rightarrow (P_2 \text{ says } Z)$. The former means that a process on a machine knows the current contents of a register on that machine; the latter means that a process on a different machine must also know the current contents of the register. But according to Necessitation, if $\vdash Z$ then $\vdash P_1 \text{ says } Z$ and $\vdash P_2 \text{ says } Z$. Only if R is guaranteed to be constant—i.e., it can never at any time be anything other than zero—must both processes say that R is zero.

Unit, therefore, is appropriate when propositions (or relations or functions) represent global state upon which all principals are guaranteed to agree. But when propositions represent local state that could be unknown to some principals, Unit would arguably be an invalid axiom. A countermodel demonstrating Unit's invalidity is easy to construct—for example, stipulate a world w at which Z holds, and let P_1's worldview contain Z but P_2's worldview not contain Z. That countermodel doesn't apply to Necessitation, because Z is not a theorem in it, therefore the principals may disagree on Z's validity.

Prior work has objected to Unit for other reasons, but not for this difference between local and global state. We are unaware of any authorization logic that rejects Necessitation, which is widely accepted along with axiom K (cf. Section 3.4) in *normal modal logic* [Hughes and Cresswell, 1996].

FOCAL was designed to reason about state in distributed systems, where prin-

cipals (such as machines) may have local state, and where global state does not necessarily exist—the reading at a clock, for example, is not agreed upon by all principals. So Unit would be invalid for FOCAL principals; Necessitation is the appropriate choice. We therefore included Necessitation in FOCAL in the form of rule SAYS-LRI, which is equivalent to Necessitation plus K [Hughes and Cresswell, 1996, p. 214, where SAYS-LRI is called LR].

Similarly, NAL principals do not necessarily agree upon global state. NAL does include Necessitation as an inference rule and does not include Unit as an axiom. However, NAL permits Unit to be derived as a theorem by the following proof:[14]

$$\text{NAL-Imp-I}_1\ \dfrac{\text{NAL-Says-I}\ \dfrac{[\varphi]^1}{p\ \mathsf{says}\ \varphi}}{\varphi \Rightarrow p\ \mathsf{says}\ \varphi}$$

NAL's proof system is, therefore, arguably unsound with respect to the belief semantics presented here: there is a formula (Unit) that is a theorem of the system but that is not semantically valid.

One way to remedy NAL's unsoundness with respect to our semantics would be to adjust our semantics, such that Unit becomes valid:

Definition 1 (U1)**.** In our belief semantics, we require that whenever $w \models \varphi$, it must hold that $\varphi \in \omega(w, p, v)$.[15]

(An equivalent condition could be imposed on the Kripke semantics.) But we chose not to do this because we want to model principals who may be ignorant

[14]Rules NAL-IMP-I and NAL-SAYS-I can be found in Schneider et al. [2011]. The brackets around φ at the top of the proof tree indicate that it is used as a hypothesis [van Dalen, 2004]. The appearance of "1" as a super- and subscript indicate where the hypothesis is introduced and cancelled.

[15]U1 was omitted from NAL [Schneider et al., 2011]. But for the NAL proof system to be sound with respect to the informal NAL belief semantics, the condition should have been imposed.

of whether certain facts hold at a world. Indeed, in our semantics, if φ holds at a world, some principals might believe φ at that world and some might not. The adjustments above would instead cause all principals to believe φ at the world, and we find this to be an unacceptable loss in expressivity.

Another way to remedy NAL's unsoundness with respect to our semantics would be to adjust NAL's proof system, such that Unit is no longer derivable. For example, a side-condition could be added to NAL-SAYS-I, such that φ must be a validity [Schneider, 2013]. One way of accomplishing that might be to forbid uncancelled hypotheses in the derivation of φ. That would prevent the above derivation of Unit, although we don't know what effect it would have on the completeness of the proof system.

FOCAL's proof system (specifically, rule SAYS-LRI) instead prohibits derivation of Unit: Unit is invalid in our semantics, and our proof system is sound with respect to our semantics, so it's impossible for our proof system to derive Unit. FOCAL therefore seems appropriate for reasoning about distributed state.

3.4.2 Soundness

Our first soundness theorem for FOCAL states that if φ is provable from assumptions Γ, and that if a belief model validates all the formulae in Γ, then that model must also validate φ. Therefore, any provable formula is valid in the belief semantics:

Theorem 10. *If* $\Gamma \vdash \varphi$ *and* $B, w, v \models \Gamma$*, then* $B, w, v \models \varphi$*.*

We have mechanized the proof of this theorem in Coq [Hirsch and Clarkson, 2013b].

The result is, to our knowledge, the first proof of soundness for an authorization logic with respect to a belief semantics. The proof of Theorem 10 relies on the following lemma, which states monotonicity of validity with respect to $\leq$:

Lemma 4. *If* $B, w, v \models \varphi$ *and* $w \leq w'$ *then* $B, w', v \models \varphi$.

Proof. By structural induction on φ. This proof has been mechanized in Coq [Hirsch and Clarkson, 2013b]. □

Our second soundness theorem for FOCAL states that any provable formula is valid in the Kripke semantics:

Theorem 11. *If* $\Gamma \vdash \varphi$ *and* $K, w, v \models \Gamma$*, then* $K, w, v \models \varphi$.

The proof of that theorem relies on Lemma 2 (monotonicity of the Kripke semantics). We also have mechanized the proofs of Theorem 11 and Lemma 2 in Coq [Hirsch and Clarkson, 2013b].

CHAPTER 4

FIRST-ORDER LOGIC FOR FLOW-LIMITED AUTHORIZATION

Distributed systems often make authorization decisions based on private data, which a public decision might leak. Preventing such leakage requires nontrivial reasoning about the interaction between information flow and authorization policies [Arden and Myers, 2016; Arden et al., 2015; Becker, 2010]. In particular, the justification for an authorization decision can violate information-flow policies. To understand this concern, consider a social network where Bob can say that only his friends may view his photos, and that furthermore only his friends may know the contents of his friend list. Assuming that there is no other way for Bob to block access to the photo, if Alice is not on Bob's friend list and she tries to view one of his photos, telling her that she does not have permission to view the photo leaks Bob's private information, specifically that Alice is not on Bob's friend list.

Reasoning about the interaction between information flow and authorization policies is challenging for several reasons. First, authorization logics use a different notion of trust from information-flow systems. Information-flow systems tend to focus on tracking data dependencies by representing information-security policies as *labels* on data. They then represent trust as a *flows-to* relation between labels, which determines when one piece of data may safely influence another. In contrast, authorization logics tend to directly encode *delegations* between principals as a *speaks-for* relation. Such delegations are often all-or-nothing, where a delegating principal trusts any statements made by the trusted principal, although some logics (e.g., [Becker et al., 2010; Howell and Kotz, 2000; Schneider et al., 2011]) support restricting delegations to specific statements. Flows-to relations implicitly encode delegations while speaks-for relations implicitly encode permitted flows. To

understand *how*, we must understand how these disparate notions of trust interact.

Both forms of trust serve to selectively constrain the communication that system components rely on to make secure authorization decisions. For example, in the social network example above, suppose Bob's security settings are recorded on server X, and his photos are stored on server Y. When Alice tries to view Bob's photo, server Y communicates with server X to determine if Alice is permitted to do so. Modeling this communication is important because (1) the servers that Y communicates with influence its authorization decisions, and (2) communication can leak private information.

Describing the information security of authorization decisions such as the one above requires modifying typical authorization policies to include information flow. Information-flow systems are excellent at tracking when and what information one principal communicates to another, specifically by transferring data from one label to another. It is less clear when communications occur in authorization logics. A common approach [Abadi, 2006; Lampson et al., 1991; Schneider et al., 2011] simply models Alice delegating trust to Bob as Alice importing all of Bob's beliefs.

On the other hand, Authorization logics *do* excel at reasoning about beliefs. Authorization logics allow us to write $\mathsf{Alice\ says}\ \varphi$, which means that Alice believes formula φ. Because this says statement is itself a formula, we can reason about what Bob believes Alice believes by nesting says formulae. Information flow, in contrast, has no notion of belief, and so cannot reason about principals' beliefs about each others' beliefs.

In order to express authorization policies, not only does one need the ability to express trust and communication, but also a battery of propositions and logical

connectives. Any tool that combines authorization and information flow should be capable of expressing enough logical connectives to reason about real-world policies. First-order logic seems to be a sweet spot of expressive power: it can encode most authorization policies, but it is still simple enough to have clean semantics. For instance, Nexus [Schneider et al., 2011; Sirer et al., 2011]—a distributed operating system that uses authorization logic directly in its authorization mechanism—can encode all of its authorization policies using first-order logic.[1]

Finally, evaluating any attempt to combine authorization and information flow policies must examine the resulting security guarantees. Both authorization logics and information-flow systems have a security property called *non-interference*. Information-flow systems view non-interference as standard, while authorization logics often view it as desirable but unobtainable. Although the two formulations look quite different, both make guarantees limiting how one component of a system can influence—i.e., interfere with—another. In authorization logics, this takes the form "Alice's beliefs can only impact the provability of Bob's beliefs if Bob trusts Alice." In information-flow systems—which are mostly defined over programs—changing the value of an input variable x can only change the value of an output variable y when the label of x flows to the label of y.

Both of these notions of non-interference are important. Consider again the example where Bob's friend list is private but Alice attempts to view his photo. Because Bob's friend list is private, changing the list should not affect Alice's beliefs. For instance, Alice should not be affected by Bob adding or removing Cathy. To enforce this, whether or not Cathy is Bob's friend must not affect the set of Bob's beliefs that Alice *may* learn. This requires authorization-logic

[1]The Nexus Authorization Logic is actually a monadic second-order logic, but this is used only to encode speaksfor and False; their examples only use first-order quantification [Schneider et al., 2011].

non-interference, since Bob's beliefs should not affect Alice's beliefs unless they communicate. It also, however, requires information-flow non-interference, since the privacy of Bob's belief is why he is unwilling to communicate.

Gluing together both ideas of non-interference requires understanding the connection between their notions of trust. As we have discussed, this connection is difficult to formulate, making the non-interference combination harder still.

Our goal in this work is to provide a logic that supports reasoning about both information flow and authorization policies by combining their models of trust to obtain the advantages of both. To this end, we present the *Flow-Limited Authorization First-Order Logic* (FLAFOL), which

- provides a notion of trust between principals that can vary depending on information-flow labels,
- clearly denotes points where communication occurs,
- uses says formulae to reason about principals' beliefs, including their beliefs about others' beliefs,
- is expressive enough to encode real-world authorization policies, and
- provides a strong security guarantee which combines both authorization-logic and information-flow non-interference.

We additionally aim to clarify the foundations of flow-limited authorization (introduced by Arden et al. [2015]). We therefore strive to keep FLAFOL's model of principals, labels, and communication as simple as possible. For example, unlike previous work, we do not require that labels form a lattice.

A final contribution is development of an implementation of FLAFOL in the

Coq proof assistant [The Coq development team, 2004] and formal proofs of all theorems in this chapter except those in Section 4.6. Together these consists of 18,384 lines of Coq code in 23 files. We hope that this will form the basis of work using FLAFOL to verify new and existing authorization mechanisms.

We are, of course, not the first to recognize the interaction of information-flow policies with authorization, but all prior work in this area is missing at least one important feature. The three projects that have done the most to combine authorization and information flow are FLAM [Arden et al., 2015], SecPAL$^+$ [Becker, 2010; Becker et al., 2010], and Aura [Jia and Zdancewic, 2009; Jia et al., 2008]. FLAM models trust using information flow, Aura uses DCC [Abadi, 2006; Abadi et al., 1999], a propositional authorization logic, and SecPAL$^+$ places information flow labels on principal-based trust policies, but does not attempt to reason about the combination at all. Neither FLAM nor SecPAL$^+$ can reason about nested beliefs, and both are significantly restricted in what logical forms are allowed. Finally, FLAM's security guarantees are non-standard and difficult to compare to other languages, while Aura relies on DCC's non-interference guarantee which does not apply on any trust relationships outside of those assumed in the static lattice.

The rest of this chapter is organized as follows: In Section 4.1 we discuss three running examples. This also serves as an intuitive introduction to FLAFOL. In Section 4.2 we use a more-detailed example to show how FLAFOL's parameterization allows it to model real systems. In Section 4.3 we detail the FLAFOL proof rules. In Section 4.4 we discuss the proof theory of FLAFOL, proving important meta-level theorems, including consistency and cut elimination. Finally, in Section 4.5 we provide FLAFOL's non-interference theorem.

4.1 FLAFOL By Example

We now examine several examples of authorization policies and how FLAFOL expresses them. This will serve as a gentle introduction to the main ideas of FLAFOL, and introduce notation and running examples we use throughout the paper.

We explore three main examples in this section:

1. Viewing pictures on social media
2. Sanitizing data inputs to prevent SQL injection attacks
3. Providing a hospital bill in the presence of reinsurance

Each setting has different requirements; for instance, each defines the meaning of labels in its own way. The ability of FLAFOL to adapt to each demonstrates its expressive power. In a new setting, it is often convenient—even necessary—to define constants, functions, and relations beyond those baked into FLAFOL. These are straightforward to define since FLAFOL's security guarantee holds for any parameterization. We use such symbols freely in our examples to express our intent clearly. Formally, FLAFOL interprets them using standard proof-theoretic techniques, as we see in Section 4.2.

Notably, FLAFOL does not allow computation on terms, so the meaning of functions and constants are axiomatized via FLAFOL formulae. This allows principals to disagree on how functions behave, which can be useful in modeling situations where each principal has their own view of some piece of data.

4.1.1 Viewing Pictures on Social Media

We begin by reconsidering in more detail the example from the beginning of this chapter where Alice requests to view Bob's picture on a social-media service. This service allows Bob to set privacy policies, and Bob made his pictures visible only to his friends. When Alice makes her request, the service can check if she is authorized by scanning Bob's friend list. If she is on the list and the photo is available, it shows her the photo. If she is *not* on Bob's friend list, it shows her an HTTP 403: Forbidden page.

Bob may choose who belongs in the role of "friend." Following the lead of other authorization logics, FLAFOL represents Bob believing that Alice is his friend as $\mathsf{Bob\ says\ IsFriend(Alice)}$. Since says statements can encompass any formula, we can express the fact that Bob believes that Alice is *not* his friend as $\mathsf{Bob\ says\ \neg IsFriend(Alice)}$.

We interpret these statements as Bob's *beliefs*. This reflects the fact that Bob could be wrong, in the sense that he may affirm formulae with provable negations. There is no requirement that Bob believes all true things nor that Bob only believe true things (see Section 4.3), so holding an incorrect belief does not require Bob to believe False. Note that because False allows us to prove anything, a principal who *does* believe False will affirm every statement.

Now imagine that, as in the beginning of this chapter, the social-media service allows Bob to set a privacy policy on his friend list as well. As before, Bob can restrict his friend list so that only his friends may learn its contents. If Alice makes her request and she is on Bob's friend list, she may again see the photo. However, if she is not, showing her an HTTP 403 page would leak Bob's private information;

Alice would learn that she is not on Bob's friend list, something Bob only shared with his friends. There is nothing the server can do but hang.

In order to discuss this in FLAFOL, we need a way to express that Bob's friend list is private. Since, formally, his friend list is a series of beliefs about who his friends are, we must express the privacy of those beliefs. We view this as giving each belief a *label* describing Bob's policy about who may learn that belief.

Syntactically, we attach this label to the says connective. For example, Bob may use the label $\mathsf{Friends}$ to represent the information-security policy "I will share this with only my friends." If he attaches this policy to the beliefs representing his friend list, there is no way to securely prove either $\mathsf{Bob\ says}_\ell\ \mathsf{IsFriend}(\mathsf{Alice})$ or $\mathsf{Bob\ says}_\ell\ \neg\mathsf{IsFriend}(\mathsf{Alice})$ when ℓ is less restrictive than $\mathsf{Friends}$; FLAFOL's security guarantee (Theorem 19) shows that every FLAFOL proof is secure, so neither option is provable in FLAFOL. Because the system searches for one proof to show Alice the picture, and the other to show her the 403 page, hanging is its only option.

A better design of the social-media service might reject policies where the only secure behavior is to hang. Such a system would only allow policies $P(\vec{x})$ where it can either prove $P(\vec{t})$ or prove $\neg P(\vec{t})$ for any list of terms $\vec{t}$. That is, the predicate P must be decidable. Since FLAFOL is intuitionistic, this is the same as a FLAFOL proof that $\forall \vec{x}, P(\vec{x}) \vee \neg P(\vec{x})$.

To avoid hanging in response to Alice's request, the social-media service thus needs the predicate $\mathsf{Bob\ says}_\ell\ \mathsf{IsFriend}(\mathsf{Alice})$ to be decidable at some ℓ that Alice can read. Unfortunately, both options leak information about Bob's friend list—which is restricted to $\mathsf{Friends}$—and all FLAFOL proofs are secure, so it must be

undecidable. If Bob's friend list were public, simply checking the list would be enough to decide this predicate. FLAFOL can easily express this by labeling each of Bob's beliefs about IsFriend as Public.

A more subtle change would be to say that every principal can find out whether *they* are on Bob's friend list, but only Bob's friends can see the rest of the list. FLAFOL can also express this policy and prove it decidable, but doing so would require significant infrastructure using the technology we will build in Sections 4.2 and 4.3.

4.1.2 Integrity Tracking to Prevent SQL Injection

For our second example, imagine a stateful web application. It takes requests, updates its database, and returns web pages. In order to avoid SQL injection attacks, the system will only update its database based on high-integrity input. However, it marks all web request inputs as low integrity, representing the fact that they may contain attacks. The server has a sanitization function San that will neutralize attacks, so when it encounters a low-integrity input, it is willing to sanitize that input and endorse the result.

FLAFOL's support for arbitrary implications allows it to easily encode such endorsements. Let the predicate $\mathsf{DBInput}(x)$ mean that a value x—possibly taken from a web request—is a database input. When a user makes a request with database input x, we can thus represent it as $\mathsf{System\ says}_{\mathsf{LInt}}\ \mathsf{DBInput}(x)$. Here LInt is a label which represents low-integrity beliefs. We represent the system's willingness to endorse any sanitized input as:

$$\mathsf{System\ says}_{\mathsf{LInt}}\ \mathsf{DBInput}(x) \rightarrow \mathsf{System\ says}_{\mathsf{HInt}}\ \mathsf{DBInput}(\mathsf{San}(x))$$

This example shows the power of arbitrary implications for expressing authorization and information-flow policies. It also, however, demonstrates their dangers, since unconstrained downgrades can allow information to flow in unintended ways. In Section 4.5 we will discuss how our non-interference theorem (Theorem 19) adapts to these downgrades by weakening its guarantees.

4.1.3 Hospital Bills Calculation and Reinsurance

Imagine now that Alice finds herself in the hospital. Luckily she has employer-provided insurance, but her employer just switched insurance companies. Now she has two unexpired insurance cards, and she cannot figure out which one should be paying for this operation. Thus, either of the two insurers I_1 and I_2 might be paying for the operation.

Imagine further that Bob's job is to create a correct hospital bill for Alice. He uses the label ℓ_H to determine both who may learn the contents of Alice's bill and who may help determine them. That is, ℓ_H expresses both a confidentiality policy and an integrity policy. Bob believes that Alice's insurer may help determine the contents of Alice's bill, since they can decide how much they are willing to pay for Alice's surgery.

Bob knows that I_2 has a reinsurance treaty with I_1. This means that if Alice is insured with I_2 and the surgery is very expensive, I_1 will pay some of the bill. Thus, I_1 may help determine the contents of Alice's hospital bill, even if I_2 turns out to be her current insurer.

Bob is willing to accept Alice's insurance cards as evidence that she is insured by either I_1 or I_2, which we express as $\mathsf{Bob\ says}_{\ell_H}\ (\mathsf{CanWrite}(I_1, \ell_H) \vee \mathsf{CanWrite}(I_2, \ell_H))$.

Because Bob knows about I_2's reinsurance treaty with I_1, he knows that if I_2 helps determine the contents of Alice's bill, they will delegate some of their power to I_1, which we express as $\mathsf{Bob\ says}_{\ell_H}\ (I_2\ \mathsf{says}_{\ell_H}\ \mathsf{CanWrite}(I_1, \ell_H))$.

Bob's beliefs allow him to prove that I_1 may help determine the contents of Alice's bill, since by assuming the previous two statements we can prove that $\mathsf{Bob\ says}_{\ell_H}\ \mathsf{CanWrite}(I_1, \ell_H)$. There are two possible cases: if Bob already believes that I_1 can help determine the contents of Alice's bill, we are done. Otherwise, Bob believes that I_2 can help determine the contents of Alice's bill, and so Bob is willing to let I_2 delegate their power. Since he knows that they will delegate their power to I_1, he knows that I_1 can help determine the contents of Alice's bill in this case as well. This covers all of the cases, so we can conclude that $\mathsf{Bob\ says}_{\ell_H}\ \mathsf{CanWrite}(I_1, \ell_H)$.

We think of Bob as performing this proof, since it is entirely about Bob's beliefs. From this point of view, Bob's ability to reason about I_2's beliefs appears to be Bob *simulating* I_2. This ability of one principal to simulate another provides the key intuition to understand the *generalized principal*, a fundamental construct in the formal presentation of FLAFOL (see Section 4.2).

We also note that Bob used I_2's beliefs in this proof, even though he does not necessarily trust I_2. However, he *might* trust it if it turns out to be Alice's insurer. Because Bob trusts I_2 in part of the proof but not in general, we refer to this as *discoverable trust*. FLAFOL's ability to handle discoverable trust makes reasoning about its security properties more difficult, as we see in Section 4.5.

This example shows how disjunctions can be used to express policies when principals do not know the state of the world. It also demonstrates how disjunctions

make it difficult to know how information can flow at any point in time, since we may discover new statements of trust under one branch of a disjunction. FLAFOL's non-interference theorem adapts to this by considering all declarations of trust that could possibly be discovered in a given context.

4.1.4 Further Adapting FLAFOL

All of the above examples use information-flow labels to express confidentiality policies, integrity policies, or both. While confidentiality and integrity are mainstay features of information flow tracking, information-flow labels can also express other properties. For instance, MixT [Milano and Myers, 2018] describes how to use information-flow labels to create safe transactions across databases with different consistency models, and the work of Zheng and Myers [2005] uses information-flow labels to provide availability guarantees. FLAFOL allows such alternative interpretations of labels by using an abstract *permission model* to give meaning to labels.

By default, the permissions gain meaning only through their behavior in context, but they are able to encode and reason about a wide variety of authorization mechanisms. In Section 4.2, we see how FLAFOL can be used to reason about capabilities in particular.

4.2 Using FLAFOL

In this section, we examine how to use FLAFOL to reason about real systems. To do this, we look at a fictional verified distributed-systems designer Dana. She wants

to formally prove that confused-deputy attacks are impossible in her capability-based system with copyable, delegatable read capabilities. Dana employs a six-step process to reason about her system in FLAFOL:

1. Decide on a set $\mathcal{S}$ of *sorts* of data she wants to represent.
2. Choose a set $\mathcal{F}$ of *function symbols* representing operations in the system, and give those operations types.
3. Choose a set $\mathcal{R}$ of *relation symbols* representing atomic facts to reason about, and give those relations types.
4. Develop axioms that encode meaning for these relationships.
5. Specify meta-level theorems stating her desired properties.
6. Prove that those meta-level theorems hold.

Sorts First, Dana decides on what sorts of data she wants to represent. We can think of *sort* as the logic word for "type." FLAFOL is defined with respect to a set $\mathcal{S}$ of sorts that must include at least Label and Principal, but may contain more. Dana wants to reason about capability tokens that grant read access to data, so she also includes a sort named Token.

Dana uses the Principal sort to represent system principals, but conceptually divides the Label sort into Confidentiality and Integrity. Each Confidentiality value defines a confidentiality policy which may be applied to many pieces of data. A capability (which is always public itself) grants read access to data governed by one or more such policies. She uses the Integrity sort to represent integrity policies on tokens themselves. We will see below how she can enforce Label = Confidentiality × Integrity.

Function Symbols Dana next decides on operations she wants to reason about. This is also her chance to define constants using nullary operations. Formally, FLAFOL is defined with respect to an arbitrary set $\mathcal{F}$ of *function symbols*. Each function comes equipped with a *signature*, or type, expressing when it can be applied.

Dana thinks about what information she needs to access about a given token. She needs to be able to determine the confidentiality a token grants permission to read, the integrity of that token, and which principal is the token's *root of authority*—that is, who created the token. She thus creates three function symbols, $\mathsf{TknConf} : \mathsf{Token} \to \mathsf{Confidentiality}$ which represents the confidentiality that a token protects, $\mathsf{IntegOfTkn} : \mathsf{Token} \to \mathsf{Integrity}$ which represents the integrity of the token, and $\mathsf{RootOfAuth} : \mathsf{Token} \to \mathsf{Principal}$ which represents who created the token. She also needs to be able to determine the integrity that a principal commands, so she includes a function symbol $\mathsf{IntegOf} : \mathsf{Principal} \to \mathsf{Integrity}$. Finally, since a token can potentially be transferred to anyone in her system, she creates a constant $\mathsf{Public} : \mathsf{Confidentiality}$ to represent this.

Dana wants to enforce that labels are pairs of confidentiality and integrity. She therefore creates two "projection" function symbols $\pi_{\mathsf{C}} : \mathsf{Label} \to \mathsf{Confidentiality}$ and $\pi_{\mathsf{I}} : \mathsf{Label} \to \mathsf{Integrity}$, along with a representation of pairing, written using mix-fix notation as $(\_, \_) : \mathsf{Confidentiality} \to \mathsf{Integrity} \to \mathsf{Label}$. The first two ensure that labels contain a confidentiality and an integrity, while pairing allows creation of labels from a confidentiality with an integrity. This makes labels pairs of confidentiality and integrity.[2]

[2]Technically, Dana also needs to write axioms equivalent to the η and β laws for pairs in order to labels to truly be pairs. However, this is a technical point that does not add to the current discussion, so we elide them below.

Relation Symbols Dana can now choose relations representing facts that she wants to reason about. Along with sorts and functions, FLAFOL is defined with respect to a set $\mathcal{R}$ of *relation symbols*, allowing it to reason about more facts. The set $\mathcal{R}$ must include at least $\mathsf{Label} \sqsubseteq \mathsf{Label}$, $\mathsf{CanRead}(\mathsf{Principal}, \mathsf{Label})$, and $\mathsf{CanWrite}(\mathsf{Principal}, \mathsf{Label})$, but may contain more. We call these required relations *permissions* because they define the trust relationships governing communication. The relation $\ell \sqsubseteq \ell'$ means information with label ℓ can affect information with label ℓ', $\mathsf{CanRead}(p, \ell)$ means that principal p may learn beliefs with label ℓ, and $\mathsf{CanWrite}(p, \ell)$ means p may influence beliefs with label ℓ.

Dana is able to use these relations to define the permissions her capability tokens grant. She also includes a fourth relation in $\mathcal{R}$, $\mathsf{HasToken}(\mathsf{Principal}, \mathsf{Token})$, defining token possession: if $\mathsf{HasToken}(p, t)$, then principal p has (a copy of) token t.

Axioms Dana describes the behavior of her system with axioms that use the sorts, functions, and relations she defined above. These should be *consistent*, in the sense that they do not allow a derivation of False. Theorem 12 in Section 4.4.1 gives conditions under which all of the axioms that we will discuss in this section are consistent.

Dana uses three main axioms: one describing how tokens may be copied and delegated, one describing when one principal may read another's beliefs, and one describing when a principal may affect another's beliefs. She may use more axioms if she likes—e.g., to capture principals' beliefs about permitted flows between labels.

Dana's first axiom allows any principal to copy any capability it holds and give

that copy to another principal:

$$\forall q : \mathsf{Principal}.\, \forall t : \mathsf{Token}.$$
$$\left(\begin{array}{l} \exists p : \mathsf{Principal}.\, \mathsf{HasToken}(p, t) \wedge \\ \\ p \ \mathsf{says}_{(\mathsf{Public}, \mathsf{IntegOfTkn}(t))} \ \mathsf{HasToken}(q, t) \end{array} \right) \rightarrow \mathsf{HasToken}(q, t)$$

This says that, for principals p and q, if p holds a read capability token t, p can pass t to q. To do so, p must affirm that q has t at a public label with the integrity of the token. The public label requires Dana to ensure that, when one principal copies a token and passes it to another principal, everyone is allowed to know this information.

Dana's second axiom defines when a principal p allows q to read a belief of p's labeled ℓ. First, p checks that q has a token, and that p believes that the token gives read access to something at least as confidential as ℓ. Second, p checks to make sure that the token's root authority may influence this belief:

$$\forall q : \mathsf{Principal}.\, \forall \ell : \mathsf{Label}.\, \forall p : \mathsf{Principal}.\, \forall \ell' : \mathsf{Label}.$$
$$\left(\begin{array}{l} \exists t : \mathsf{Token}.\, \mathsf{HasToken}(q, t) \\ \qquad\qquad \wedge\ p \ \mathsf{says}_{\ell'} \ \pi_{\mathsf{C}}(\ell) \sqsubseteq \mathsf{TknConf}(t) \\ \qquad\qquad \wedge\ p \ \mathsf{says}_{\ell'} \ \mathsf{CanWrite}(\mathsf{RootOfAuth}(t), \ell') \end{array} \right)$$
$$\rightarrow p \ \mathsf{says}_{\ell'} \ (\mathsf{CanRead}(q, \ell))$$

More formally, it says that if q holds some token t and p believes both that t grants read permissions for ℓ's confidentiality and that the root of authority for t can influence p's beliefs at ℓ', then p will allow q to read ℓ. This defines what it means for a principal (p here) to believe that a token grants read access to their data. Dana now needs to make sure that whenever a read access is granted in her system, not only does the principal who gets read access have a token, but that the principal who owns the data does indeed believe that the token grants read access to that data.

Sorts	σ	::=	$\mathsf{Label} \mid \mathsf{Principal} \mid \cdots$
Labels	ℓ		
Principals	p, q, r		
Functions	f	::=	$\cdots$
Relations	R	::=	$\mathsf{CanRead}(\mathsf{Principal}, \mathsf{Label})$
		$\mid$	$\mathsf{CanWrite}(\mathsf{Principal}, \mathsf{Label})$
		$\mid$	$\mathsf{Label} \sqsubseteq \mathsf{Label} \mid \cdots$
σ-terms	t	::=	$x \mid f(t_1, \ldots, t_n)$
Formulae	φ, ψ, χ	::=	$R(t_1, \ldots, t_n)$
		$\mid$	$\mathsf{True} \mid \mathsf{False}$
		$\mid$	$\varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \to \psi$
		$\mid$	$\forall x\!:\!\sigma.\, \varphi \mid \exists x\!:\!\sigma.\, \varphi$
		$\mid$	$p\ \mathsf{says}_\ell\ \varphi$
Generalized Principals	g	::=	$\langle\rangle \mid g \cdot p\langle\ell\rangle$

Figure 4.1: FLAFOL Syntax

Finally, her third axiom states that a principal can write a label (according to a second principal) if the integrity of that principal flows to the integrity of the label:

$$\forall q\!:\!\mathsf{Principal}.\, \forall \ell\!:\!\mathsf{Label}.\, \forall p\!:\!\mathsf{Principal}.\, \forall \ell'\!:\!\mathsf{Label}.$$
$$p\ \mathsf{says}_{\ell'}\ (\mathsf{IntegOf}(q) \sqsubseteq \pi_{\mathsf{I}}(\ell)) \to p\ \mathsf{says}_{\ell'}\ (\mathsf{CanWrite}(q, \ell))$$

Metatheoretic Properties Dana has now created a model of her system, so she can use it to state and prove properties of her system as meta-theorems. Luckily, Rajani et al. [2016] have shown that information-flow integrity tracking with a non-interference result is sufficient to avoid confused deputy attacks with capability systems. Therefore Theorem 19 provides the guarantees she needs.

FLAFOL Syntax This example demonstrates FLAFOL's flexibility as a powerful tool for reasoning about authorization mechanisms with information-flow policies. We saw that, since FLAFOL is defined with respect to the three sets $\mathcal{S}$, $\mathcal{F}$,

and $\mathcal{R}$, it can express the key components of a system. This parameterized definition gives rise to the formal FLAFOL syntax in Figure 4.1.

In order to use the function and relation symbols and incorporate axioms, FLAFOL allows proofs to occur in a context. FLAFOL additionally includes rules requiring flows-to to be reflexive and transitive, placing a preorder on the Label sort,[3] and requiring CanRead and CanWrite to respect a form of variance. If $\ell_1 \sqsubseteq \ell_2$ and Alice can read data A with label ℓ_2, then she may learn information about data with label ℓ_1 used to calculate A. This means she should also be able to read data with label ℓ_1. Thus, CanRead must (contravariantly) respect the preorder on labels. Similarly, if Alice can help determine some piece of data B labeled with ℓ_1, she can influence any data labeled with ℓ_2 that is calculated from B, so Alice should be able to help determine data labeled at ℓ_2. Thus, CanWrite must (covariantly) respect the preorder on labels.

Figure 4.2 presents these rules formally. We give the proof rules in the form of a sequent calculus. The trailing @ g represents *who* affirms that formula in the proof, similarly to how says formulae represent who affirms a statement at the object level. Unlike says formulae, these meta-level objects—called *generalized principals*—encode arbitrary reasoners, including possibly-simulated principals.

Recall from Section 4.1.3 that we can think of some proofs as being performed by principals if those proofs entirely involve that principal's beliefs. In that example, Bob reasoned about his belief that another principal, the insurer I_2, trusted a third principal, the insurer I_3. We think of this ability to reason about the beliefs of others as the ability to *simulate* other principals. In fact, because principals'

[3]Many information-flow tools require their labels to form a lattice. We find that a preorder is sufficient for FLAFOL's design and guarantees, so we decline to impose additional structure. In Section 4.4.1 we show that enforcing a lattice structure is both simple and logically consistent.

$$\textsc{FlowsToRefl}\ \frac{}{\Gamma \vdash \ell \sqsubseteq \ell \mathbin{@} g}$$

$$\textsc{FlowsToTrans}\ \frac{\Gamma \vdash \ell_1 \sqsubseteq \ell_2 \mathbin{@} g \qquad \Gamma \vdash \ell_2 \sqsubseteq \ell_3 \mathbin{@} g}{\Gamma \vdash \ell_1 \sqsubseteq \ell_3 \mathbin{@} g}$$

$$\textsc{CRVar}\ \frac{\Gamma \vdash \mathsf{CanRead}(p, \ell_2) \mathbin{@} g \qquad \Gamma \vdash \ell_1 \sqsubseteq \ell_2 \mathbin{@} g}{\Gamma \vdash \mathsf{CanRead}(p, \ell_1) \mathbin{@} g}$$

$$\textsc{CWVar}\ \frac{\Gamma \vdash \mathsf{CanWrite}(p, \ell_1) \mathbin{@} g \qquad \Gamma \vdash \ell_1 \sqsubseteq \ell_2 \mathbin{@} g}{\Gamma \vdash \mathsf{CanWrite}(p, \ell_2) \mathbin{@} g}$$

Figure 4.2: Permission Rules

beliefs are segmented by labels, principals can have multiple simulations of the same other principal.

This suggests that FLAFOL captures the reasoning of principals *at some level of simulation*. A generalized principal is a stack of principal/label pairs, representing a stack of simulators and simulations. The empty stack, written $\langle\rangle$, represents *ground truth*. A stack with one more level, written $g \cdot p\langle\ell\rangle$, represents the beliefs of p at level ℓ according to the generalized principal g. Figure 4.1 contains the formal grammar for generalized principals.

4.3 Proof System

So far, we have discussed the intuitions behind FLAFOL and its syntax. Here we introduce FLAFOL formally. Unfortunately, we cannot examine every aspect of FLAFOL's formal presentation in detail, though interested readers should see Appendix B. Instead, we discuss the most novel and most security-relevant aspects of FLAFOL's design.

FLAFOL sequents are of the form $\Gamma \vdash \varphi \mathbin{@} g$, where Γ is a context containing beliefs. This means that the FLAFOL proof system manipulates beliefs, as described in Section 4.2. Readers familiar with sequent calculus may recognize that FLAFOL is intuitionistic, as there is only one belief on the right side of the turnstile.

Sequent calculus rules tend to manipulate beliefs either on the left or the right side of the turnstile. For instance, consider the FLAFOL rules for conjunctions:

$$\textsc{AndL}\ \frac{\Gamma, \varphi \mathbin{@} g, \psi \mathbin{@} g \vdash \chi \mathbin{@} g'}{\Gamma, (\varphi \wedge \psi \mathbin{@} g) \vdash \chi \mathbin{@} g'} \qquad \textsc{AndR}\ \frac{\Gamma \vdash \varphi \mathbin{@} g \qquad \Gamma \vdash \psi \mathbin{@} g}{\Gamma \vdash \varphi \wedge \psi \mathbin{@} g}$$

We find it easiest to read left rules "up" and right rules "down." With this reading, the AndL rule uses an assumption of the form $\varphi \wedge \psi \mathbin{@} g$ by splitting it into two assumptions, one for each conjunct, while the AndR rule takes proofs of two formulae and proves their conjunction.[4]

Most of the rules of FLAFOL are standard rules for first-order logic with generalized principals included to indicate who believes each formula. For instance, the rules for conjunctions above were likely familiar to those who know sequent calculus.

Figure 4.3 contains FLAFOL rules selected for discussion. The first, FalseL, tells us how to use False as an assumption. In standard intuitionistic first-order logic, this is simply the principle of Ex Falso: if we assume False, we can prove anything. In FLAFOL, a generalized principal who assumes false is willing to affirm any formula. This includes statements about other principals, so FalseL extends the generalized principal arbitrarily. We use $g \cdot g'$ as notation for extending

[4]For readers interested in learning more about sequent calculus, we recommend MIT's interactive tool for teaching sequent caluclus as a tutorial [Yang, 2012].

$$\textsc{FalseL}\ \frac{}{\Gamma, \mathsf{False} \mathbin{@} g \vdash \varphi \mathbin{@} g \cdot g'}$$

$$\textsc{OrL}\ \frac{\Gamma, \varphi \mathbin{@} g \vdash \chi \mathbin{@} g' \qquad \Gamma, \psi \mathbin{@} g \vdash \chi \mathbin{@} g'}{\Gamma, (\varphi \vee \psi \mathbin{@} g) \vdash \chi \mathbin{@} g'} \qquad \textsc{OrR1}\ \frac{\Gamma \vdash \varphi \mathbin{@} g}{\Gamma \vdash \varphi \vee \psi \mathbin{@} g}$$

$$\textsc{OrR2}\ \frac{\Gamma \vdash \psi \mathbin{@} g}{\Gamma \vdash \varphi \vee \psi \mathbin{@} g}$$

$$\textsc{ImpL}\ \frac{\Gamma \vdash \varphi \mathbin{@} \langle\rangle \qquad \Gamma, \psi \mathbin{@} g \vdash \chi \mathbin{@} g'}{\Gamma, (\varphi \to \psi \mathbin{@} g) \vdash \chi \mathbin{@} g'} \qquad \textsc{ImpR}\ \frac{\Gamma, \varphi \mathbin{@} \langle\rangle \vdash \psi \mathbin{@} g}{\Gamma \vdash \varphi \to \psi \mathbin{@} g}$$

$$\textsc{SaysL}\ \frac{\Gamma, \varphi \mathbin{@} g \cdot p\langle\ell\rangle \vdash \psi \mathbin{@} g'}{\Gamma, p\ \mathsf{says}_\ell\ \varphi \mathbin{@} g \vdash \psi \mathbin{@} g'} \qquad \textsc{SaysR}\ \frac{\Gamma \vdash \varphi \mathbin{@} g \cdot p\langle\ell\rangle}{\Gamma \vdash p\ \mathsf{says}_\ell\ \varphi \mathbin{@} g}$$

$$\textsc{VarR}\ \frac{\Gamma \vdash \varphi \mathbin{@} g \cdot p\langle\ell'\rangle \cdot g' \qquad \Gamma \vdash \ell' \sqsubseteq \ell \mathbin{@} g \cdot p\langle\ell\rangle}{\Gamma \vdash \varphi \mathbin{@} g \cdot p\langle\ell\rangle \cdot g'}$$

$$\textsc{FwdR}\ \frac{\begin{array}{c}\Gamma \vdash \varphi \mathbin{@} g \cdot p\langle\ell\rangle \cdot g' \\ \Gamma \vdash \mathsf{CanRead}(q, \ell) \mathbin{@} g \cdot p\langle\ell\rangle \qquad \Gamma \vdash \mathsf{CanWrite}(p, \ell) \mathbin{@} g \cdot q\langle\ell\rangle\end{array}}{\Gamma \vdash \varphi \mathbin{@} g \cdot q\langle\ell\rangle \cdot g'}$$

Figure 4.3: Selected FLAFOL Proof Rules

the generalized principal g with a list of principal-label pairs, denoted g'.

We discuss the disjunction rules OrR1, OrR2, and OrL because says distributes over disjunctions. That is, given that $p\ \mathsf{says}_\ell\ (\varphi \vee \psi)$, we can prove that $(p\ \mathsf{says}_\ell\ \varphi) \vee (p\ \mathsf{says}_\ell\ \psi)$. In an intuitionistic logic like FLAFOL,[5] disjunctions must be a proof of one side or the other. The proof of distribution of says over $\vee$ then says that if p has evidence of either φ or ψ, then p can examine this evidence to discover whether it is evidence of φ or evidence of ψ. However, it does mean there are some situations that we cannot model, such as one where Alice knows that Bob has a hidden bit, but does not know whether it is on or off.

[5]Recall that we argued in Section 4.1.1 that reasoning about authorization and information-flow security together is naturally intuitionistic.

The implication rules ImpR and ImpL interpret the premise of an implication as ground truth, while the generalized principal who believes the implication believes the consequent. In particular, this means that says statements do not distribute over implication as one might expect, i.e., $p \ \mathsf{says}_\ell \ (\varphi \to \psi)$ does not imply that $(p \ \mathsf{says}_\ell \ \varphi) \to (p \ \mathsf{says}_\ell \ \psi)$. Instead, $p \ \mathsf{says}_\ell \ (\varphi \to \psi)$ implies $\varphi \to (p \ \mathsf{says}_\ell \ \psi)$. We can thus think of implications as *conditional* knowledge. That is, if a generalized principal g believes $\varphi \to \psi$, then g believes ψ conditional on φ being true about the system.

We can still form implications about generalized principals' beliefs, but we must insert appropriate says statements into the premise to do so. In Section 4.4.6, we discuss how this semantics is necessary for both our proof theoretic and security results.

The next two rules of Figure 4.3, SaysR and SaysL, are the only rules which specifically manipulate says formulae. Essentially, generalized principals allow us to delete the says part of a formula while not forgetting who said it. Thus, generalized principals allow us to define sequent calculus rules once for every possible reasoner.

The final rules, VarR and FwdR, define communication in FLAFOL. Both manipulate beliefs on the right and have corresponding left rules which act contravariantly. Those left rules can be found in Appendix B.

Information-flow communication is provided by the variance rule VarR. This can be thought of like the variance rules used in subtyping. Most systems with information-flow labels do not have explicit variance rules, but instead manipulate relevant labels in every rule. By adding an explicit variance rule, we not only simplify every other FLAFOL rule, we also remove the need for the label join and

meet operators that are usually used to perform the label manipulations. Others have noted that adding explicit variance rules improves the design of the rest of the system [Algehed, 2018; Volpano et al., 1996], but it remains an unusual choice.

The forwarding rule FWD provides authorization-logic-style communication. In FLAFOL, p can forward a belief at label ℓ to q if:

- p is willing to send its beliefs at label ℓ to q, denoted $p \ \mathsf{says}_\ell \ \mathsf{CanRead}(q, \ell)$, and
- q is willing to allow p to determine its beliefs at label ℓ, which we write $q \ \mathsf{says}_\ell \ \mathsf{CanWrite}(p, \ell)$.

After establishing this trust, p can package up its belief and send it to q, who will believe it at the same label.

4.4 Proof Theory

In this section, we evaluate FLAFOL's logical design. We show that FLAFOL has the standard sequent calculus properties of (left) consistency, (positive) consistency and cut elimination and discuss fundamental limitations that inform our unusual implication semantics. We also develop a new proof-theoretic tool, *compatible supercontexts*, for use in our non-interference theorem in Section 4.5. Finally, we evaluate FLAFOL's design as a *modal* logic by showing how it admits (a version of) the classic proof rule for modalities.

$$s \in \{+,-\} \qquad \overline{+} = - \qquad \overline{-} = +$$

$$\frac{}{\varphi^s \leq \varphi^s} \qquad \frac{\varphi^s \leq \psi^{s'} \qquad \psi^{s'} \leq \chi^{s''}}{\varphi^s \leq \chi^{s''}} \qquad \frac{}{\varphi^s \leq (\varphi \vee \psi)^s} \qquad \frac{}{\psi^s \leq (\varphi \vee \psi)^s}$$

$$\frac{}{\varphi^s \leq (\varphi \wedge \psi)^s} \qquad \frac{}{\psi^s \leq (\varphi \wedge \psi)^s} \qquad \frac{}{\varphi^{\overline{s}} \leq (\varphi \to \psi)^s} \qquad \frac{}{\psi^s \leq (\varphi \to \psi)^s}$$

$$\frac{}{(\varphi[x \mapsto t])^- \leq (\forall x\!:\!\sigma.\, \varphi)^-} \qquad \frac{}{\varphi^+ \leq (\forall x\!:\!\sigma.\, \varphi)^+} \qquad \frac{}{\varphi^- \leq (\exists x\!:\!\sigma.\, \varphi)^-}$$

$$\frac{}{(\varphi[x \mapsto t])^+ \leq (\exists x\!:\!\sigma.\, \varphi)^+} \qquad \frac{}{\varphi^s \leq (p \;\mathsf{says}_\ell\; \varphi)^s}$$

Figure 4.4: Signed Subformula Relation

4.4.1 Consistency

One of the most important properties about a logic is consistency, meaning it is impossible to prove False. This is not possible in an arbitrary context, since one could always assume False. One standard solution is to limit the theorem to the empty context. By examining the FLAFOL proof rules, however, we see that it is only possible to prove False by assumption or by Ex Falso. Either method requires that False already be on the left-hand side of the turnstile, so if False can never get there, then it should be impossible to prove.

To understand when False can appear on the left-hand side of the turnstile, we note that formulae on the left tend to stay on the left and formulae on the right tend to stay on the right. The only exception is the implication rules ImpL and ImpR which move the premise of the implication to the other side. The fact that no proof rule allows us to arbitrarily change either side of the sequent gives useful structure to proofs. To handle implications, however, we must keep track of their nesting structure, which we do by considering *signed formulae*. We call a formula in a sequent *positive* if it appears on the right side of the turnstile and *negative* if it

appears on the left. If φ is positive we write φ^+, and if φ is negative we write φ^-. Figure 4.4 shows rules defining the *signed subformula relation*, which we discuss in more depth in Section 4.4.2.

The intuition above and this relation lead to the following theorem. Note that formulae which do not contain False as a negative subformula are called *positive* formulae, explaining the name.

Theorem 12 (Positive Consistency)**.** *For any context* Γ*, if*

$$\mathsf{False}^- \not\leq \varphi^- \text{ for all } \varphi \mathbin{@} g \in \Gamma$$

then $\Gamma \nvdash \mathsf{False} \mathbin{@} g'$.

The proof follows by induction on the FLAFOL proof rules.

We get the result with an empty context as a corollary. This states that False is not a theorem of FLAFOL.

Corollary 1 (Consistency)**.** $\nvdash \mathsf{False} \mathbin{@} g$

Proof. Because Γ is empty, the "for all" premise in Theorem 12 is vacuously true. □

Theorem 12 demonstrates that a variety of useful constructs are logically consistent. For instance, we can add a lattice structure to FLAFOL's labels. We can define join ($\sqcup$) and meet ($\sqcap$) as binary function symbols on labels and $\top$ and $\bot$ as label constants. Then we can simply place the lattice axioms (e.g., $\forall \ell : \mathsf{Label}.\ \ell \sqsubseteq \top$) in our context to achieve the desired result. Since none of the lattice axioms include False, Theorem 12 ensures that they are consistent additions to the logic.

4.4.2 Signed Subformula Property

As we mention in Section 4.4.1, FALFOL formulae tend not to move between the left-hand side of the turnstile and the right-hand side. Moreover, the only exceptions to this are the implication rules. This means that looking at where a subformula appears in a sequent tells us which side of the turnstile it can appear on for the rest of the proof.

Note that every subformula of a signed formula has a unique sign. If a subformula appears by itself in a sequent during a proof, then which side of the turnstile it is on is determined by its sign. This structure results in the following formal property.

Theorem 13 (Left Signed-Subformula Property)**.** *If* $\Gamma \vdash \varphi \mathbin{@} g_1$ *appears in a proof of* $\Delta \vdash \psi \mathbin{@} g_2$*, then for all* $\chi_1 \mathbin{@} g_3 \in \Gamma$*, either (1)* $\chi_1^- \leq \psi^+$ *or (2) there is some* $\chi_2 \mathbin{@} g_4 \in \Delta$ *such that* $\chi_1^- \leq \chi_2^-$*.*

The proof follows by induction on the FLAFOL proof rules.

Many logics also have a similar *right* signed-subformula property. FLAFOL does not enjoy that property since $\Gamma \vdash \varphi \mathbin{@} g_1$ may be a side condition on a forward or a variance rule, and thus not related directly to ψ.

4.4.3 Compatible Supercontexts

To prove Theorems 12 and 13 we needed to consider the possible locations of *formulae* within a sequent, but in Section 4.5 we will need to reason about the possible locations of *beliefs*. To enable this, we introduce the concept of a *compat-*

$$\textsc{CSCRefl}\ \frac{}{\Gamma \ll \Gamma \vdash \varphi \mathbin{@} g} \qquad \textsc{CSCUnion}\ \frac{\Delta_1 \ll \Gamma \vdash \varphi \mathbin{@} g \qquad \Delta_2 \ll \Gamma \vdash \varphi \mathbin{@} g}{\Delta_1 \cup \Delta_2 \ll \Gamma \vdash \varphi \mathbin{@} g}$$

$$\textsc{CSCOrL1}\ \frac{\Delta \ll \Gamma, \varphi \mathbin{@} g \vdash \chi \mathbin{@} g'}{\Delta \ll \Gamma, (\varphi \vee \psi \mathbin{@} g) \vdash \chi \mathbin{@} g'} \qquad \textsc{CSCImpR}\ \frac{\Delta \ll \Gamma, \varphi \mathbin{@} \langle\rangle \vdash \psi \mathbin{@} g}{\Delta \ll \Gamma \vdash \varphi \to \psi \mathbin{@} g}$$

Figure 4.5: Selected Rules for Compatible Supercontexts

ible supercontext (CSC). Informally, the CSCs of a sequent are those contexts that contain all of the information in the current context, along with any counterfactual information that can be considered during a proof. Intuitively, the rules OrL and ImpL allow a generalized principal to consider such information by using either side of a disjunction or the conclusion of an implication. If it is possible to consider such a counterfactual, there is a CSC which contains it. We use the syntax $\Delta \ll \Gamma \vdash \varphi \mathbin{@} g$ to denote that Δ is a CSC of the sequent $\Gamma \vdash \varphi \mathbin{@} g$. Figure 4.5 contains selected rules for CSCs. Others can be found in Appendix C.

Since all of the information in Γ has already been discovered by the generalized principal who believes that information, we require that $\Gamma \ll \Gamma \vdash \varphi \mathbin{@} g$ with CSCRefl.

If we can discover two sets of information, we can discover everything in the union of those sets using CSCUnion. This rule feels different from the others, since it axiomatizes certain *properties* of CSCs. We conjecture that there is an alternative presentation of CSCs for which we can prove that this rule is admissable.

The rest of the rules for CSCs essentially follow the proof rules, so that any belief added to the context during a proof can be added to a CSC. For instance CSCOrL1 and CSCOrL2 allow either branch of an assumed disjunction to be added to a CSC, following the two branches of the OrL rule of FLAFOL.

If a context appears in a proof of a sequent, then it is a CSC of that sequent. We refer to this as the *compatible-supercontext property* (CSC property).

Theorem 14 (CSC Property)**.** *If* $\Delta \vdash \psi \mathbin{@} g'$ *appears in a proof of* $\Gamma \vdash \varphi \mathbin{@} g$*, then* $\Delta \ll \Gamma \vdash \varphi \mathbin{@} g$*.*

The following lemma about compatible supercontexts helps us prove our main security result.

Lemma 5. *The following two rules are admissible up to* α *equivalence of* $\Gamma \vdash \varphi \mathbin{@} g_1$*:*

$$\textsc{CSCAtomic}\ \frac{\Delta \ll \Gamma \vdash \varphi \mathbin{@} g_1 \qquad \varphi \textit{ is atomic}}{\Delta \ll \Gamma \vdash \psi \mathbin{@} g_2}$$

$$\textsc{CSCWeaken}\ \frac{\Delta \ll \Gamma \vdash \varphi \mathbin{@} g_1}{\Delta, \psi \mathbin{@} g_2 \ll \Gamma, \psi \mathbin{@} g_2 \vdash \varphi \mathbin{@} g_1}$$

Proof. Note that this lemma is not available in the Coq. In order to avoid reasoning about α equivalence, we instead added these as CSC rules. Both proofs are by induction on the derivation of $\Delta \ll \Gamma \vdash \varphi \mathbin{@} g_1$.

CSCAtomic follows from the fact that no right rules that consider the shape of φ apply when it is atomic. CSCIR, CSCSCR, CSCVarR, and CSCFwdR can simply be eliminated when we replace $\varphi \mathbin{@} g_1$ with $\psi \mathbin{@} g_2$. All other rules either cannot apply when φ is atomic, or apply equally with $\psi \mathbin{@} g_2$.

CSCWeaken follows from a direct straightforward induction (with heavy use of CSCExchange) noting that CSCRefl is the only way to terminate the induction. □

4.4.4 Simulation

In (multi-)modal logics, we are interested in modeling *perfect* reasoners. That is, reasoners should reason correctly based on their assumed beliefs; if their assumed beliefs were true, then all of their derived beliefs would be as well.

In most logics (which do not have generalized principals), this is axiomatized as a rule in the system, written as follows:

$$\textsc{SaysF}\ \frac{\Gamma \vdash \varphi}{p\ \mathsf{says}_\ell\ \Gamma \vdash p\ \mathsf{says}_\ell\ \varphi}$$

(Note that this is essentially SaysLRI from Chapter 3. The name SaysF which we use here refers to the fact that this makes $p\ \mathsf{says}_\ell$ *functorial* for every p and ℓ.) Here, $p\ \mathsf{says}_\ell\ \Gamma$ refers to a copy of Γ with $p\ \mathsf{says}_\ell$ in front of every formula in Γ. In such logics, SaysF is the main rule for manipulating says statements. However, this requires removing all beliefs that are not those of p at level ℓ in a context before using this rule to reason as p at level ℓ.

FLAFOL instead uses the says introduction rules in Section 4.3, which allows us to retain the beliefs of other principals and of p at other labels, making it easier to discuss communication. However, FLAFOL reasoners are still perfect reasoners, which we show by proving that FLAFOL admits a rule analogous to SaysF. We refer to this as the *simulation* theorem, since it says that p is correctly "simulating the world in its head."

FLAFOL does not precisely admit SaysF for two reasons. The first is that our belief syntax pushes says statements into generalized principals, so we must place the new principal-label pair at the beginning of the generalized principal instead of on the formula. The second is that the semantics of implications in FLAFOL

mean that $p \mathsf{\ says}_\ell (\varphi \to \psi)$ has different semantics from $(p \mathsf{\ says}_\ell \varphi) \to (p \mathsf{\ says}_\ell \psi)$. To address this concern, we define the $\odot$ operator:

$$p\langle\ell\rangle \odot \varphi \triangleq \begin{cases} (p \mathsf{\ says}_\ell (p\langle\ell\rangle \odot \psi)) \to (p\langle\ell\rangle \odot \chi) & \varphi = \psi \to \chi \\ (p\langle\ell\rangle \odot \psi) \wedge (p\langle\ell\rangle \odot \chi) & \varphi = \psi \wedge \chi \\ (p\langle\ell\rangle \odot \psi) \vee (p\langle\ell\rangle \odot \chi) & \varphi = \psi \vee \chi \\ \forall x\!:\!\sigma.\, (p\langle\ell\rangle \odot \psi) & \varphi = \forall x\!:\!\sigma.\, \psi \\ \exists x\!:\!\sigma.\, (p\langle\ell\rangle \odot \psi) & \varphi = \exists x\!:\!\sigma.\, \psi \\ \varphi & \text{otherwise} \end{cases}$$

This essentially "repairs" implications to have the right says statements in front of the premise.

We allow the same syntax to prepend to a generalized principal, defining

$$p\langle\ell\rangle \odot (\langle\rangle \cdot g') \triangleq \langle\rangle \cdot p\langle\ell\rangle \cdot g'.$$

We can therefore lift the operator to beliefs straightforwardly, defining

$$p\langle\ell\rangle \odot (\varphi \mathbin{@} g) \triangleq (p\langle\ell\rangle \odot \varphi) \mathbin{@} p\langle\ell\rangle \odot g,$$

and from there to contexts as:

$$p\langle\ell\rangle \odot \Gamma \triangleq \begin{cases} \cdot & \Gamma = \cdot \\ (p\langle\ell\rangle \odot \Gamma'), p\langle\ell\rangle \odot (\varphi \mathbin{@} g) & \Gamma = \Gamma', \varphi \mathbin{@} g \end{cases}$$

With this definition in hand, we can now state the simulation theorem in full:

Theorem 15 (Simulation)**.** *The following rule is admissible:*

$$\frac{\Gamma \vdash \varphi \mathbin{@} g}{(p\langle\ell\rangle \odot \Gamma) \vdash p\langle\ell\rangle \odot (\varphi \mathbin{@} g)}$$

The proof follows by induction on the proof.

4.4.5 Cut Elimination

In constructing a proof, it is often useful to create a lemma, prove it separately, and use it in the main proof. If we both prove and use the lemma in the same context, the main proof follows in that context as well. We can formalize this via the following rule:

$$\textsc{Cut}\ \frac{\Gamma \vdash \varphi \mathbin{@} g_1 \qquad \Gamma, \varphi \mathbin{@} g_1 \vdash \psi \mathbin{@} g_2}{\Gamma \vdash \psi \mathbin{@} g_2}$$

This rule is enormously powerful. It allows us to not only create lemmata to use in a proof, but also prove things that do not obviously have other proofs. For instance, consider the rule

$$\textsc{UnsaysR}\ \frac{\Gamma \vdash p\ \mathsf{says}_\ell\ \varphi \mathbin{@} g}{\Gamma \vdash \varphi \mathbin{@} g \cdot p\langle \ell \rangle}$$

We can show that this rule is *admissible*—meaning any sequent provable with this rule is provable without it—by cutting a proof of the sequent $\Gamma \vdash p\ \mathsf{says}_\ell\ \varphi \mathbin{@} g$ with the following proof:[6]

$$\textsc{SaysL}\ \frac{\textsc{Ax}\ \dfrac{}{\varphi \mathbin{@} g \cdot p\langle \ell \rangle \vdash \varphi \mathbin{@} g \cdot p\langle \ell \rangle}}{p\ \mathsf{says}_\ell\ \varphi \mathbin{@} g \vdash \varphi \mathbin{@} g \cdot p\langle \ell \rangle}$$

However, the Cut rule allows an arbitrary formula to appear on both sides of the turnstile in a proof. That formula may not even be a subformula of anything in the sequent at the root of the proof-tree! This would seemingly destroy the CSC property that FLAFOL enjoys, and which we rely on in order to prove FLAFOL's

[6]Not only can UnsaysR be proven admissable without Cut (as can all FLAFOL proofs), it is actually important for proving cut elimination.

security results. As is standard in sequent calculus proof theory, we show that CUT can be admitted, allowing FLAFOL the proof power of CUT while maintaining the analytic power of the CSC property.

Theorem 16 (Cut Elimination)**.** *The* CUT *rule is admissible.*

This theorem is one of the key theorems of proof theory [Girard et al., 1989; Takeuti, 1987]. Pfenning [1995] has called it "[t]he central property of sequent calculi." From the propositions-as-types perspective, cut elimination is preservation of types under substitution, a fact that will be important in the next section.

The proof of Theorem 16 uses the fact that every proof can be put into a certain *normal form.* [7] A proof is in normal form if all rules which do not manipulate formulae are higher in the proof tree than those which do. Formally, we define two normal forms, called first and second normal form, which represent "might use formula-manipulating rules" and "will not use formula manipulating rules", respectively. A proof is in first normal form if, when a rule which manipulates something other than a formula is used, all subproofs above that rule are in second normal form, while a proof is in second normal form it if never uses any rules which manipulate formulae.

Theorem 17 (FLAFOL Normalization)**.** *If* $\Gamma \vdash \varphi \mathbin{@} g$ *is provable, then it has a normal-form proof.*

The proof is by induction on the proof tree of $\Gamma \vdash \varphi \mathbin{@} g$.

In order to prove 16, we first normalize both proofs. If they're both in First Normal Form but not in Second Normal Form, we proceed as suggested by Pfenning

[7] In the literature, "normal proof" refers to a cut-free proof, rather than a proof in FLAFOL's normal form.

[1995]: nested triple induction on the formula being cut and on both proofs. If one of them is in Second Normal Form we use a different procedure. This procedure consists of getting the dual rule to the last rule used in the proof that is in Second Normal Form (e.g. VarL for the VarR case) and make it the last rule to the other proof. Due to the covariant-contravariant nature of these rules and their duals, this is always possible.

4.4.6 Implications and Communication

Recall from Section 4.3 how we interpret implications such as $\mathsf{Alice\ says}_\ell\ (\varphi \to \psi)$: if φ is true about the system, then Alice knows ψ at label ℓ. We can now understand why FLAFOL uses this interpretation.

Imagine we replace rules ImpL and ImpR with rules that interpret the above formula in a more intuitive fashion: if Alice believes φ at label ℓ, then she also believes ψ at ℓ. That is, we replace ImpL and ImpR with the following rules:

$$\textsc{ImpL}'\ \frac{\Gamma \vdash \varphi\ @\ g \qquad \Gamma, \psi\ @\ g \vdash \chi\ @\ g'}{\Gamma, (\varphi \to \psi)\ @\ g \vdash \chi\ @\ g'} \qquad\qquad \textsc{ImpR}'\ \frac{\Gamma, \varphi\ @\ g \vdash \psi\ @\ g}{\Gamma \vdash \varphi \to \psi\ @\ g}$$

This would allow us to prove that says distributes over implication, as you can see in Figure 4.6: if $\mathsf{Alice\ says}_\ell\ (\varphi \to \psi)$ then $(\mathsf{Alice\ says}_\ell\ \varphi) \to (\mathsf{Alice\ says}_\ell\ \psi)$.[8] It would also, unfortunately, allow us to prove the converse, as you can see in Figure 4.7

In this setting we can provide a proof that contains a cut we cannot eliminate and demonstrates a security flaw. Imagine that Alice receives a TopSecret version of φ from Cathy, and she wants to prove ψ at TopSecret. Alice can also prove ψ publicly if she believes φ privately, but doing so requires sending φ to Bob.

[8]Note that, for space reasons, we drop $\langle\rangle\cdot$ and $@\ \langle\rangle$ from proofs in this section.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varphi \mathbin{@} g \cdot p\langle\ell\rangle \vdash \varphi \mathbin{@} g \cdot p\langle\ell\rangle}\,\textsc{Ax}}{p \mathsf{\ says}_\ell\ \varphi \mathbin{@} g \vdash \varphi \mathbin{@} g \cdot p\langle\ell\rangle}\,\textsc{SaysL} \qquad \dfrac{\dfrac{}{\psi \mathbin{@} g \cdot p\langle\ell\rangle \vdash \psi \mathbin{@} g \cdot p\langle\ell\rangle}\,\textsc{Ax}}{\psi \mathbin{@} g \cdot p\langle\ell\rangle \vdash p \mathsf{\ says}_\ell\ \psi \mathbin{@} g}\,\textsc{SaysR}}{(\varphi \to \psi) \mathbin{@} g \cdot p\langle\ell\rangle, p \mathsf{\ says}_\ell\ \varphi \mathbin{@} g \vdash p \mathsf{\ says}_\ell\ \psi \mathbin{@} g}\,\textsc{ImpL}'}{(\varphi \to \psi) \mathbin{@} g \cdot p\langle\ell\rangle \vdash (p \mathsf{\ says}_\ell\ \varphi) \to (p \mathsf{\ says}_\ell\ \psi) \mathbin{@} g}\,\textsc{ImpR}'}{p \mathsf{\ says}_\ell\ (\varphi \to \psi) \mathbin{@} g \vdash (p \mathsf{\ says}_\ell\ \varphi) \to (p \mathsf{\ says}_\ell\ \psi) \mathbin{@} g}\,\textsc{SaysL}$$

Figure 4.6: Proof that ImpL′ and ImpR′ allow says to distribute over implication.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varphi \mathbin{@} g \cdot p\langle\ell\rangle \vdash \varphi \mathbin{@} g \cdot p\langle\ell\rangle}\,\textsc{Ax}}{\varphi \mathbin{@} g \cdot p\langle\ell\rangle \vdash p \mathsf{\ says}_\ell\ \varphi \mathbin{@} g}\,\textsc{SaysR} \qquad \dfrac{\dfrac{}{\psi \mathbin{@} g \cdot p\langle\ell\rangle \vdash \psi \mathbin{@} g \cdot p\langle\ell\rangle}\,\textsc{Ax}}{p \mathsf{\ says}_\ell\ \psi \mathbin{@} g \vdash \psi \mathbin{@} g \cdot p\langle\ell\rangle}\,\textsc{SaysL}}{(p \mathsf{\ says}_\ell\ \varphi) \to (p \mathsf{\ says}_\ell\ \psi) \mathbin{@} g, \varphi \mathbin{@} g \cdot p\langle\ell\rangle \vdash \psi \mathbin{@} g \cdot p\langle\ell\rangle}\,\textsc{ImpL}'}{(p \mathsf{\ says}_\ell\ \varphi) \to (p \mathsf{\ says}_\ell\ \psi) \mathbin{@} g \vdash (\varphi \to \psi) \mathbin{@} g \cdot p\langle\ell\rangle}\,\textsc{ImpR}'}{(p \mathsf{\ says}_\ell\ \varphi) \to (p \mathsf{\ says}_\ell\ \psi) \mathbin{@} g \vdash p \mathsf{\ says}_\ell\ (\varphi \to \psi) \mathbin{@} g}\,\textsc{SaysR}$$

Figure 4.7: Proof that ImpL′ and ImpR′ allow says to undistribute over implication.

By packaging this proof into an implication using these new rules and then using variance, we obtain a proof that, if Alice believes φ at TopSecret, she can prove ψ at TopSecret.[9] We can cut these two proofs together, but eliminating this cut would force Alice or Cathy to send a TopSecret belief to Bob, though they are unwilling.

This insecurity stems from the ability to both distribute and *un-distribute* says across implications while also using the variance and forward rules. Adopting a propositions-as-types viewpoint provides further insight. In this perspective, the says modalities are type constructors, the variance and forwarding rules act as subtyping relations on the resulting types, and implications are functions. The forward and variance rules require functions to behave *contravariantly* on their inputs, as normal. Also allowing says to distribute over implications in both directions, however, would force functions to behave *covariantly* on their inputs. A standard type-theoretic argument suggests that this makes β-reduction—i.e., cut

[9]Note that ImpL and ImpR do not allow this proof.

elimination—impossible. By treating premises as ground truth, functions become *invariant* on their premises, allowing us to prove cut elimination for FLAFOL.

Let us consider our example in more detail. There are three principals of interest: Alice, Bob, and Cathy, and three labels: ℓ_0, ℓ_1, and ℓ_2, representing Public, $\mathsf{Private}$, and $\mathsf{TopSecret}$, respectively. (We use the shorter names for the sake of readability of formal proofs.) Anybody in the system can read public data (i.e., data labeled with ℓ_0). Alice and Cathy believe all three principals of interest can read private data (i.e., data labeled with ℓ_1), but Bob is unsure of the security clearances and will only send public data to other principals. Alice and Cathy also have top secret clearance, but Bob does not, so he *cannot* read data labeled at ℓ_2. Finally, Bob serves as a redactor: given φ—which represents a document containing private information—he can produce ψ—which represents a redacted version of the same document—performing a declassification in the process.

The information needed to formalize these permission policies in our proof are in the context below:

$$\begin{aligned}\Gamma = {} & \forall p{:}\mathsf{Principal}.\, p\ \mathsf{says}_{\ell_1}\ \ell_0 \sqsubseteq \ell_1,\\ & \forall p{:}\mathsf{Principal}.\, p\ \mathsf{says}_{\ell_2}\ \ell_1 \sqsubseteq \ell_2,\\ & \mathsf{CanRead}(\mathsf{Bob}, \ell_1)\ @\ \mathsf{Alice}\langle \ell_1 \rangle,\\ & \mathsf{CanRead}(\mathsf{Alice}, \ell_2)\ @\ \mathsf{Cathy}\langle \ell_2 \rangle,\\ & \forall p, q{:}\mathsf{Principal}.\, p\ \mathsf{says}_{\ell_0}\ \mathsf{CanRead}(q, \ell_0),\\ & \forall p, q{:}\mathsf{Principal}.\, \forall \ell, \ell'{:}\mathsf{Label}.\, p\ \mathsf{says}_{\ell}\ \mathsf{CanWrite}(q, \ell')\end{aligned}$$

To represent Bob's ability to redact information from the document φ, we add one additional belief:

$$\Gamma' = \Gamma, (\mathsf{Bob}\ \mathsf{says}_{\ell_1}\ \varphi) \to (\mathsf{Bob}\ \mathsf{says}_{\ell_0}\ \psi)$$

$$\textsc{ImpL}'\ \dfrac{\textsc{Ax}\ \dfrac{}{\Gamma, \varphi \mathbin{@} \mathsf{Alice}\langle \ell_2 \rangle \vdash \varphi \mathbin{@} \mathsf{Alice}\langle \ell_2 \rangle} \qquad \textsc{Ax}\ \dfrac{}{\Gamma, \varphi \mathbin{@} \mathsf{Alice}\langle \ell_2 \rangle, \psi \mathbin{@} \mathsf{Alice}\langle \ell_2 \rangle \vdash \psi \mathbin{@} \mathsf{Alice}\langle \ell_2 \rangle}}{\textsc{FwdL}^\dagger\ \dfrac{\Gamma, (\varphi \to \psi) \mathbin{@} \mathsf{Alice}\langle \ell_2 \rangle, \varphi \mathbin{@} \mathsf{Alice}\langle \ell_2 \rangle \vdash \psi \mathbin{@} \mathsf{Alice}\langle \ell_2 \rangle}{\Gamma, (\varphi \to \psi) \mathbin{@} \mathsf{Alice}\langle \ell_2 \rangle, \varphi \mathbin{@} \mathsf{Cathy}\langle \ell_2 \rangle \vdash \psi \mathbin{@} \mathsf{Alice}\langle \ell_2 \rangle}}$$

Figure 4.8: Alice using Cathy's φ and a redaction function

Imagine further that Alice decides she wants to redact private information from a TopSecret version of φ that she receives from Cathy, but leave it TopSecret. If she can figure out how to get a redaction implication, she'll simply receive φ from Cathy and then use the implication. This is the proof in Figure 4.8. Note that, for the sake of brevity and readability, we do not explicitly state side conditions which are proven straightforwardly from Γ. The rules where these side conditions should appear are marked with "$\dagger$."

While she knows how to *use* and implication representing redaction, Alice does not know how to redact φ except by giving it to Bob. Using ImpL$'$ and ImpR$'$, she is able to package up the process "give Bob a secret version of φ, get back a public version of ψ, and then use variance to get a private version of ψ" as a belief $\varphi \to \psi \mathbin{@} \mathsf{Alice}\langle \ell_1 \rangle$. She can then use variance again to get a belief $\varphi \to \psi \mathbin{@} \mathsf{Alice}\langle \ell_2 \rangle$. This is the proof in Figure 4.9. Again, we elide side conditions which are proved straightforwardly from Γ, and mark the rules where they should appear with "$\dagger$."

Cutting these two proofs together gives Alice what she wants: a TopSecret version of ψ. However, this cut is not possible to eliminate! Examining this through a propositions-as-types lens tells us why: one of Alice or Cathy must send a TopSecret version of φ to Bob, which neither is unwilling to do.

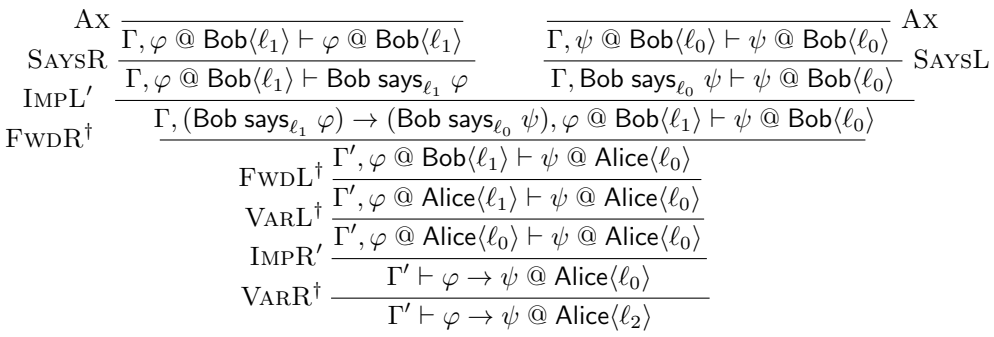


Figure 4.9: Proof corresponding to Alice sending φ to Bob and receiving a ψ back

4.5 Non-Interference

Both authorization logics and information flow systems have security properties called *non-interference* [Denning, 1976; Garg and Pfenning, 2006; Goguen and Meseguer, 1982], both of which are considered important. On the face, these two notions of non-interference look very different, but their core intuitions are the same. Both statements aim to prevent one belief or piece of data from interfering with another—*even indirectly*—unless the security policies permit an influence. Authorization logics traditionally define trust relationships between principals and non-interference requires that p's beliefs affect the provability of q's beliefs only when q trusts p. Information flow control systems generally specify policies as labels on program data and use the label flows-to relation to constrain how inputs can affect outputs. For non-interference to hold, changing an input with label ℓ_1 can only alter an output with label ℓ_2 if $\ell_1 \sqsubseteq \ell_2$.

FLAFOL views both trust between principals and flows between labels as ways to constrain communication of beliefs. The forward rules model an authorization-logic-style sending of beliefs from one principal to another based on their trust relationships. The label variance rules model a single principal transferring beliefs

between labels based on the flow relationship between them. By reasoning about generalized principals, which include both the principal and the label, we are able to capture both at the same time. The result (Theorem 19) mirrors the structure of existing authorization logic non-interference statements [Abadi, 2006; Garg and Pfenning, 2006]. No similar theorem reasons about information flow or applies to policies combining discoverable trust and logical disjunction. Theorem 19 does both.

4.5.1 Trust in FLAFOL

Building a notion of trust on generalized principals requires us to consider both the trust of the underlying (regular) principals and label flows. The explicit label flow relation ($\sqsubseteq$) cleanly captures restrictions on changing labels. Trust between principals requires more care. Alice may trust Bob with public data, but that does not mean she trusts him with secret data. Similarly, Alice may believe that Bob can influence low integrity data without believing Bob is authorized to influence high integrity data. This need to trust principals differently at different labels leads us to define our trust in terms of the two permission relations: $\mathsf{CanRead}(p, \ell)$ and $\mathsf{CanWrite}(p, \ell)$.

We group label flows and principal trust together in a meta-level statement relating generalized principals. As this relation is the fundamental notion of trust in FLAFOL, we follow existing authorization logic literature and call it *speaks for*.

The speaks-for relation captures any way that one generalized principal's beliefs can be safely transfered to another. This can happen through flow relationships ($g \cdot p\langle\ell\rangle$ speaks for $g \cdot p\langle\ell'\rangle$ if $\ell \sqsubseteq \ell'$), forwarding ($g \cdot p\langle\ell\rangle$ speaks for $g \cdot q\langle\ell\rangle$ if p can

$$\textsc{ReflSF}\ \frac{}{\Gamma \vdash g\ \mathsf{SF}\ g} \qquad \textsc{ExtSF}\ \frac{\Gamma \vdash g_1\ \mathsf{SF}\ g_2}{\Gamma \vdash g_1 \cdot p\langle\ell\rangle\ \mathsf{SF}\ g_2 \cdot p\langle\ell\rangle}$$

$$\textsc{SelfLSF}\ \frac{}{\Gamma \vdash g \cdot p\langle\ell\rangle\ \mathsf{SF}\ g \cdot p\langle\ell\rangle \cdot p\langle\ell\rangle} \qquad \textsc{SelfRSF}\ \frac{}{\Gamma \vdash g \cdot p\langle\ell\rangle \cdot p\langle\ell\rangle\ \mathsf{SF}\ g \cdot p\langle\ell\rangle}$$

$$\textsc{VarSF}\ \frac{\Gamma \vdash \ell \sqsubseteq \ell'\ @\ g \cdot p\langle\ell'\rangle}{\Gamma \vdash g \cdot p\langle\ell\rangle\ \mathsf{SF}\ g \cdot p\langle\ell'\rangle}$$

$$\textsc{FwdSF}\ \frac{\Gamma \vdash \mathsf{CanRead}(q, \ell)\ @\ g \cdot p\langle\ell\rangle \qquad \Gamma \vdash \mathsf{CanWrite}(p, \ell)\ @\ g \cdot q\langle\ell\rangle}{\Gamma \vdash g \cdot p\langle\ell\rangle\ \mathsf{SF}\ g \cdot q\langle\ell\rangle}$$

$$\textsc{TransSF}\ \frac{\Gamma \vdash g_1\ \mathsf{SF}\ g_2 \qquad \Gamma \vdash g_2\ \mathsf{SF}\ g_3}{\Gamma \vdash g_1\ \mathsf{SF}\ g_3}$$

Figure 4.10: The rules defining speaks for.

forward beliefs at ℓ to q), and introspection ($g \cdot p\langle\ell\rangle$ speaks for $g \cdot p\langle\ell\rangle \cdot p\langle\ell\rangle$ and vice versa). We formalize speaks-for with the rules in Figure 4.10.

To validate this notion of trust, we note that existing authorization logics often define speaks-for as an atomic relation and create trust by requiring that, if p speaks for q, then p's beliefs can be transfered to q. As our speaks-for relation exactly mirrors FLAFOL's rules for communication, it enjoys this same property.

Theorem 18 (Speaks-For Elimination)**.** *FLAFOL admits the following rule:*

$$\textsc{ElimSF}\ \frac{\Gamma \vdash \varphi\ @\ g_1 \qquad \Gamma \vdash g_1\ \mathsf{SF}\ g_2}{\Gamma \vdash \varphi\ @\ g_2}$$

This notion of trust allows us to begin structuring a non-interference statement. We might like to say that beliefs of g_1 can only influence beliefs of g_2 if $\Gamma \vdash g_1\ \mathsf{SF}\ g_2$. Formally, this might take the form: if $\Gamma, (\varphi\ @\ g_1) \vdash \psi\ @\ g_2$ is provable, then either $\Gamma \vdash \psi\ @\ g_2$ is provable or $\Gamma \vdash g_1\ \mathsf{SF}\ g_2$. Unfortunately, this statement is false for three critical reasons: says statements, implication, and the combination of

discoverable trust and disjunctions.

4.5.2 Says Statements and Non-Interference

The first way to break the proposed non-interference statement above is simply by moving affirmations of a statement between the formula—using says—and the generalized principal who believes it. For example, we can trivially prove $p\ \mathsf{says}_\ell\ \varphi\ @\ \langle\rangle \vdash \varphi\ @\ \langle\rangle \cdot p\langle\ell\rangle$, yet we cannot prove $\langle\rangle\ \mathsf{SF}\ \langle\rangle \cdot p\langle\ell\rangle$.

To address this case, we can view $p\ \mathsf{says}_\ell\ \varphi\ @\ \langle\rangle$ as a statement that $\langle\rangle \cdot p\langle\ell\rangle$ believes φ. This insight suggests that we might generally push all says modalities into the generalized principal. We can do this for simple formulae, but the process breaks down with conjunction and disjunction. In those cases, the different sides may have different says modalities, and either side could influence a belief through the different resulting generalized principals. We alleviate this concern by considering a *set* of generalized principals referenced in a given belief. We build this set using an operator $\mathcal{G}$:

$$\mathcal{G}(\chi\ @\ g) \triangleq \begin{cases} \mathcal{G}(\varphi\ @\ g \cdot p\langle\ell\rangle) & \chi = p\ \mathsf{says}_\ell\ \varphi \\ \mathcal{G}(\varphi\ @\ g) \cup \mathcal{G}(\psi\ @\ g) & \chi = \varphi \wedge \psi \text{ or } \varphi \vee \psi \\ \mathcal{G}(\psi\ @\ g) & \chi = \varphi \to \psi \\ \bigcup_{t:\sigma} \mathcal{G}(\varphi[x \mapsto t]\ @\ g) & \chi = \forall x{:}\sigma.\, \varphi \text{ or } \exists x{:}\sigma.\, \varphi \\ \{g\} & \text{otherwise} \end{cases}$$

For implications, $\mathcal{G}$ only considers the consequent, as the implication cannot affect the provability of a belief unless its consequent can. For quantified formulae, a proof may substitute any term of the correct sort for the bound variable, so we must as well.

Using this new operator, we can patch the hole says statements created in our previous non-interference statement, producing the following: If $\Gamma, (\varphi \mathbin{@} g_1) \vdash \psi \mathbin{@} g_2$, then either $\Gamma \vdash \psi \mathbin{@} g_2$, or there is some $g_1' \in \mathcal{G}(\varphi \mathbin{@} g_1)$, $g_2' \in \mathcal{G}(\psi \mathbin{@} g_2)$, and some g_1'' such that $\Gamma \vdash g_1' \cdot g_1'' \mathsf{SF} g_2'$.

Here g_1'' represents the ability of a generalized principal to ship entire simulations to other generalized principals. In particular, the forward and variance rules operate on an "active" prefix of the current generalized principal; g_1'' represents the suffix.

The $\mathcal{G}$ operator converts reasoning about beliefs from the object level (FLAFOL formulae) to the meta level (generalized principals). FLAFOL's ability to freely move between the two forces us to push all such reasoning in the same direction to effectively compare the reasoner in two different beliefs. Prior authorization logics do not contain a meta-level version of says, meaning similar conversions do not even make sense.

4.5.3 Implications

While use of the $\mathcal{G}$ function solves part of the problem with our original non-interference proposal, it does not address all of the problems. Implications can implicitly create new trust relationships, allowing beliefs of one generalized principal to affect beliefs of another, even when no speaks-for relationship exists. To understand how this can occur, we revisit our example of preventing SQL injection attacks from from Section 4.1.2.

Recall from Section 4.1.2 that a web server might treat sanitized versions of low-integrity input as high integrity. Further recall, it might represent this willingness

with the following implication.

$$\mathsf{System\ says}_{\mathsf{LInt}}\ \mathsf{DBInput}(x) \rightarrow \mathsf{System\ says}_{\mathsf{HInt}}\ \mathsf{DBInput}(\mathsf{San}(x))$$

In an intuitively-sensible context where System believes $\mathsf{HInt} \sqsubseteq \mathsf{LInt}$—high integrity flows to low integrity—but not vice versa, there is no way to prove $\mathsf{System}\langle\mathsf{LInt}\rangle\ \mathsf{SF}\ \mathsf{System}\langle\mathsf{HInt}\rangle$. The presence of this implication, however, allows some beliefs at $\mathsf{System}\langle\mathsf{LInt}\rangle$ to influence beliefs at $\mathsf{System}\langle\mathsf{HInt}\rangle$. This influence is actually an endorsement from LInt to HInt, and our speaks-for relation explicitly does not capture such effects.

Prior work manages this trust-creating effect of implications either by claiming security only when all implications are provable [Abadi, 2006] or by explicitly using assumed implications to represent trust [Garg and Pfenning, 2006]. We hew closer to the latter model and make the implicit trust of implications explicit in our statement of non-interference. We therefore cannot use the speaks-for relation, so we construct a new relation between generalized principals we call *can influence*.

Intuitively, g_1 can influence g_2—denoted $\Gamma \vdash g_1\ \mathsf{CanInfl}\ g_2$—if either g_1 speaks for g_2 or there is an implication in Γ that allows a belief of g_1 to affect the provability of a belief of g_2. This relation, formally defined in Figure 4.11, uses the $\mathcal{G}$ operator discussed above to capture the generalized principals actually discussed by each subformla of the implication. Because FLAFOL interprets the premise of an implication as a condition whose modality is independent of the entire belief, so too does the can-influence relation. The relation is also transitive, allowing it to capture the fact that a proof may require many steps to go from a belief at g_1 to a belief at g_2.

Simply taking our attempted non-interference statement from above and re-

$$\text{SF-CI}\ \frac{\Gamma \vdash g_1\ \mathsf{SF}\ g_2}{\Gamma \vdash g_1\ \mathsf{CanInfl}\ g_2} \qquad \textsc{ExtCI}\ \frac{\Gamma \vdash g_1\ \mathsf{CanInfl}\ g_2}{\Gamma \vdash g_1 \cdot g'\ \mathsf{CanInfl}\ g_2 \cdot g'}$$

$$\textsc{TransCI}\ \frac{\Gamma \vdash g_1\ \mathsf{CanInfl}\ g_2 \qquad \Gamma \vdash g_2\ \mathsf{CanInfl}\ g_3}{\Gamma \vdash g_1\ \mathsf{CanInfl}\ g_3}$$

$$\textsc{ImpCI}\ \frac{\varphi \to \psi\ @\ g \in \Gamma \qquad g_1 \in \mathcal{G}(\varphi\ @\ \langle\rangle) \qquad g_2 \in \mathcal{G}(\psi\ @\ g)}{\Gamma \vdash g_1\ \mathsf{CanInfl}\ g_2}$$

Figure 4.11: The rules defining the *can influence* relation.

placing speaks-for with can-influence allows us to straightforwardly capture the effect of implications on trust within the system.

While this change may appear small, it results in a highly conservative estimate of possible influence. Implications are precise statements that can allow usually-disallowed information flows under very particular circumstances. Unfortunately, because our non-interference statement only considers the generalized principals involved, not the entire beliefs, it cannot represent the same level of precision. A single precise implication added to a context can therefore relate whole classes of previously-unrelated generalized principals, eliminating the ability for non-interference to say anything about their relative security.

This same lack of precision in information flow non-interference statements has resulted in long lines of research on how to precisely model or safely restrict declassification and endorsement [Askarov and Myers, 2011; Cecchetti et al., 2017; Chong and Myers, 2008; Li and Zdancewic, 2005; Mantel and Sands, 2004; Myers et al., 2006; Sabelfeld and Myers, 2004; Sabelfeld and Sands, 2005; Waye et al., 2015; Zdancewic and Myers, 2001]. It would be interesting future work to apply these analyses and restrictions to FLAFOL to produce more precise statements of security.

4.5.4 Discovering Trust with Disjunctions

The $\mathcal{G}$ operator and can-influence relation address difficulties from both says formulae and implications, but our statement of non-interference still does not account for the combination of disjunctions and the ability to discover trust relationships. To understand the effect of these two features in combination, recall the reinsurance example from Section 4.1.3. Bob can derive $\mathsf{CanWrite}(I_1, \ell_H)$ if he already believes both $\mathsf{CanWrite}(I_1, \ell_H) \vee \mathsf{CanWrite}(I_2, \ell_H)$ and I_2 says_{ℓ_H} $\mathsf{CanWrite}(I_1, \ell_H)$.

We clearly cannot remove either of Bob's beliefs and still prove the result. Our desired theorem statement would thus require that $\mathsf{Bob}\langle\ell_H\rangle \cdot I_2\langle\ell_H\rangle$ can influence $\mathsf{Bob}\langle\ell_H\rangle$, which there is no way to prove. The reason the sequent is still provable, as we noted in Section 4.1.3, is that Bob can *discover* trust in I_2 when he branches on an Or statement, which then allows I_2 to influence Bob. In this branch, we can prove $\mathsf{Bob}\langle\ell_H\rangle \cdot I_2\langle\ell_H\rangle$ SF $\mathsf{Bob}\langle\ell_H\rangle \cdot \mathsf{Bob}\langle\ell_H\rangle$, which then speaks for $\mathsf{Bob}\langle\ell_H\rangle$.

To handle such assumptions, we cannot simply consider the context in which we are proving a sequent; we must consider any context that can appear in the proof of that sequent. We developed the notion of compatible supercontexts in Section 4.4.3 for exactly this purpose. Indeed, if we replace Γ with an appropriate CSC when checking the potential influence of generalized principals, we remove the last barrier to a true non-interference theorem.

4.5.5 Formal Non-Interference

The techniques above allow us to modify our attempted non-interference statement into a theorem that holds.

Theorem 19 (Non-Interference)**.** *For all contexts* Γ *and beliefs* $\varphi \mathbin{@} g_1$ *and* $\psi \mathbin{@} g_2$*, if* $\Gamma, \varphi \mathbin{@} g_1 \vdash \psi \mathbin{@} g_2$*, then either (1)* $\Gamma \vdash \psi \mathbin{@} g_2$*, or (2) there is some* $\Delta \ll \Gamma, \varphi \mathbin{@} g_1 \vdash \psi \mathbin{@} g_2$*,* $g_1' \in \mathcal{G}(\varphi \mathbin{@} g_1)$*,* $g_2' \in \mathcal{G}(\psi \mathbin{@} g_2)$*, and* g_1'' *such that* $\Delta \vdash g_1' \cdot g_1''$ CanInfl g_2'*.*

This follows by induction on the proof of the sequent $\Gamma, \varphi \mathbin{@} g_1 \vdash \psi \mathbin{@} g_2$. For each proof rule, we argue that either $\varphi \mathbin{@} g_1$ is unnecessary for all premises or we can extend an influence from one or more subproofs to an influence from $\varphi \mathbin{@} g_1$ to $\psi \mathbin{@} g_2$.

This theorem limits when a belief $\varphi \mathbin{@} g_1$ can be necessary to prove $\psi \mathbin{@} g_2$ in context Γ, much like other authorization logic non-interference statements [Abadi, 2006; Garg and Pfenning, 2006]. As we mentioned above, however, it is the first such non-interference statement for any authorization logic supporting all first-order connectives and discoverable trust. Moreover, it describes how FLAFOL mitigates both:

- communication between principals, through CanRead and CanWrite statements, and
- movement of information between security levels represented by information flow labels, via flows-to statements.

The CanInfl relation seems to make our non-interference statement much less precise than we would like. After all, implications precisely specify what beliefs can be declassified or endorsed, whereas CanInfl conservatively assumes any beliefs can move between the relevant generalized principals. This lack of precision serves a purpose. It allows us to reason about any implications, including those that arbitrarily change principals and labels, something which other no authorization logics

have done before. It is therefore worth noting that, when all of the implications in the context are provable, the theorem holds *even if you replace* CanInfl *with* SF *everywhere.* The same proof works, with some simple repair in the ImpL case.

Another complaint of imprecision applies to compatible supercontexts. Specifically, if any principal assumes $\varphi \vee \neg\varphi$ for any formula φ, then there is a CSC in which that principal has assumed both, even though these are arrived at through mutually-exclusive choices. Since CSCs have been added in order to allow disjunctions and discoverable trust to co-exist, it is good to know that if we disallow either, CSCs are not required for non-interference. That is, if there are no disjunctions in the context, then we can always instantiate the Δ in Theorem 19 with $\Gamma, \varphi \mathbin{@} g_1$. Similarly, if every permission that is provable under any CSC of $\Gamma, \varphi \mathbin{@} g_1 \vdash \psi \mathbin{@} g_2$ is provable under $\Gamma, \varphi \mathbin{@} g_1$, then we can again always instantiate Δ with $\Gamma, \varphi \mathbin{@} g_1$.

Together, these points demonstrate that there are only two types of poorly-behaved formulae that force the imprecision in Theorem 19. This further shows that our non-interference result is no less precise than those of other authorization logics in the absence of such formulae. We add imprecision only when needed to allow our statement to apply to more proofs. Interesting future research would allow for a more precise non-interference theorem even in the presence of such formulae.

To see how Theorem 19 corresponds to traditional non-interference results for information flow, consider a setting where every principal agrees on the same label ordering, and where there are no implications corresponding to declassifications or endorsements. Then any two contexts Γ and Γ' which disagree only on beliefs labeled above some ℓ can prove exactly the same things at label ℓ—$\Gamma \vdash \varphi \mathbin{@} g \cdot p\langle\ell\rangle$ if and only if $\Gamma' \vdash \varphi \mathbin{@} g \cdot p\langle\ell\rangle$—since Theorem 19 allows us to delete all of the

beliefs on which they disagree. If we view contexts as inputs, as in a propositions-as-types interpretation, then this says that changing high inputs cannot change low results.

4.6 Respect of Permission Beliefs

The non-interference statement of the previous section gives a powerful guarantee about security, combining both authorization-logic and information-flow notions of trust. We achieve this by stating non-interference in terms of *generalized* principals. However, when considering the security of a system, the security of (proper) *principals* matters.

In FLAFOL different principals have different information-security policies, expressed through permissions. Thus, the security of principals requires that Alice's beliefs about permissions be respected on Alice's data. For example, Alice's beliefs at label ℓ should only affect Bob's beliefs (at any label) if Alice believes $\mathsf{CanRead}(\mathsf{Bob}, \ell)$. Enforcing this is only possible if Alice exclusively sends data to people who will respect her policies. If Cathy is willing to send any data labeled ℓ to Bob, but Alice believes Cathy can read label ℓ and Bob cannot, Cathy might violate Alice's policies.

To ensure that nobody violates Alice's policies, it therefore seems necessary for Alice to only send data to principals who hold similar beliefs. Such a strong restriction, however, does not account for the fact that Alice may trust Cathy to *decide* who can read data at ℓ. Specifically, if Alice believes $\mathsf{CanWrite}(\mathsf{Cathy}, \ell)$, then Alice would accept Cathy's claim that $\mathsf{CanRead}(\mathsf{Bob}, \ell)$. This does not necessarily mean that Alice also holds this belief, because telling Alice that $\mathsf{CanRead}(\mathsf{Bob}, \ell)$

may violate *Cathy's* information-security policies. The result is a situation where Alice's permission beliefs are respected, Alice's beliefs at ℓ influence Bob's, but Alice does not believe $\mathsf{CanRead}(\mathsf{Bob}, \ell)$.

Our formal notion of trustworthiness describes these two cases explicitly. The case where Cathy must hold similar beliefs to Alice formalizes that anyone Cathy may pass Alice's beliefs to must be someone Alice believes can read that information.

Definition 2 (Trustworthy)**.** We say that a principal p is *trustworthy* with respect to q at label ℓ in world (g, Γ) if either (1) $\Gamma \vdash \mathsf{CanWrite}(p, \ell)$ @ $g \cdot q\langle\ell\rangle$, or (2) for all ℓ' such that $\Gamma \vdash \ell \sqsubseteq \ell'$ @ $g \cdot p\langle\ell'\rangle$,

- $\Gamma \vdash \ell \sqsubseteq \ell'$ @ $g \cdot q\langle\ell'\rangle$, and
- $\{r \mid \Gamma \vdash \mathsf{CanRead}(r, \ell')$ @ $g \cdot p\langle\ell'\rangle\}$
  $\subseteq \{r \mid \Gamma \vdash \mathsf{CanRead}(r, \ell')$ @ $g \cdot q\langle\ell'\rangle\}$.

In this definition, we refer to a pair of a generalized principal and a context (g, Γ) as a *world*. Since FLAFOL beliefs are held by generalized principals and must be proven in a context, we can only reason about p's beliefs surrounding permissions (or anything else) in a world simulated by a generalized principal g and defined by context Γ. For example, we can read a proof of $\Gamma \vdash \varphi$ @ g as saying that φ holds in world (g, Γ).

Definition 2 captures who will respect Alice's permission beliefs at a given label, and therefore where Alice should be willing to send data. We would like to say that Alice can safely send data to trustworthy people, but trustworthiness is unfortunately not quite sufficient. Alice may believe $\mathsf{CanWrite}(\mathsf{Cathy}, \ell)$, but Cathy may send data at ℓ to someone untrustworthy. If Cathy—and everyone she

sends data to, recursively—is trustworthy, however, *then* we can verify that Alice's permission beliefs are respected. We refer to Alice as *wise* if she will only send data to principals who will (recursively) respect her beliefs.

Definition 3 (Wisdom)**.** We say that p is *wise* at label ℓ in world (g, Γ) if for all labels ℓ' and principals q, $\Gamma \vdash \ell \sqsubseteq \ell' \mathbin{@} g \cdot p\langle\ell'\rangle$ and $\Gamma \vdash \mathsf{CanRead}(q, \ell') \mathbin{@} g \cdot p\langle\ell'\rangle$ imply that q is both trustworthy with respect to p and wise at ℓ' in world (g, Γ).

Using these definitions we can formulate how FLAFOL respects permission beliefs. Specifically, we analyze when q can receive beliefs from p, formalized using the speaks-for relation. We return to speaks-for because can-influence is expressly designed to capture the implicit trust created when assumed implications violate existing trust relationships. This makes can-influence poorly suited to making strong statements about respecting permissions.

Using speaks-for, we show that, if a principal p is wise, then any other principal q who can receive p's beliefs must acquire them through a trustworthy chain. A chain is trustworthy if each principal believes that the following principal has permission to both see the belief and decide who else can see it. The label may vary incrementally, and each hop in the chain must believe that the variance it sees is permitted by the flows-to-relation. Finally, the recipient of the data (q) must transitively trust the entire sequence to provide information at the resulting label, allowing q to safely incorporate the resulting belief. We formalize this intuition as follows.

Theorem 20 (Respect of Permission Beliefs[10])**.** *For all contexts* Γ*, generalized principals* g*, regular principals* p *and* q*, and labels* ℓ *and* ℓ'*, if* $\Gamma \vdash g \cdot p\langle\ell\rangle \mathsf{SF}\ g \cdot q\langle\ell'\rangle$ *is provable without using* ExtSF*,* SelfLSF*, or* SelfRSF *and* p *is wise at* ℓ *in*

[10]Recall that no theorems in this section have been proven in Coq.

world (g, Γ)*, then there exists a sequence* $(p_0, \ell_0), \ldots, (p_n, \ell_n)$ *such that* $p_0 = p$*,* $\ell_0 = \ell$*,* $p_n = q$*,* $\Gamma \vdash \ell_n \sqsubseteq \ell' \mathbin{@} g \cdot q\langle \ell' \rangle$*, and for all* $i \in [1, n]$*,*

- $\Gamma \vdash \ell_{i-1} \sqsubseteq \ell_i \mathbin{@} g \cdot p_{i-1}\langle \ell_i \rangle$,
- $\Gamma \vdash \mathsf{CanRead}(p_i, \ell_i) \mathbin{@} g \cdot p_{i-1}\langle \ell_i \rangle$,
- $\Gamma \vdash \mathsf{CanWrite}(p_{i-1}, \ell_i) \mathbin{@} g \cdot p_i\langle \ell_i \rangle$,
- $i = n$ *or* $\Gamma \vdash \mathsf{CanWrite}(p_i, \ell_i) \mathbin{@} g \cdot p_{i-1}\langle \ell_i \rangle$.

To prove Theorem 20, we first examine the proof that $\Gamma \vdash g \cdot p\langle \ell \rangle \mathsf{\ SF\ } g \cdot q\langle \ell' \rangle$. We construct a (potentially-longer) sequence of principal-label pairs transitioning from $p\langle \ell \rangle$ to $q\langle \ell' \rangle$, each using a flow belief or a trust belief. We can then combine flows and forwards into single steps. Using the wisdom of p at ℓ, we can also bypass forwards whose trustworthiness relies on the second case of Definition 2 rather than the first, giving us the desired trust conditions.

While our proof relies on the speaks-for proof not using ExtSF, SelfLSF, or SelfRSF, we conjecture that this condition is redundant for generalized principals of the appropriate form.

Conjecture 1. *If* $\Gamma \vdash g \cdot p\langle \ell \rangle \mathsf{\ SF\ } g \cdot q\langle \ell' \rangle$ *is provable, then it is provable without using* ExtSF*,* SelfLSF*, and* SelfRSF*.*

Theorem 20 is a novel security result for FLAFOL. It captures that principals' information-security policies can only be violated by those that they trust, transitively. Not only does it provide a security result for (proper) principals, it describes how Alice's beliefs about who can decide her policies can safely cause her data to be shared with others whom she would not be willing share with directly.

The core of this proof is contained in the following lemma.

Lemma 6. *Let $(p_0, \ell_0), \ldots, (p_m, \ell_m)$ be a sequence of principal-label pairs. Assume p_0 is wise at ℓ_0 in world (g, Γ) and, for all $i \in [1, m]$, $\Gamma \vdash \ell_{i-1} \sqsubseteq \ell_i \mathbin{@} g \cdot p_{i-1}\langle \ell_i \rangle$ and either*

1. *$p_{i-1} = p_i$, or*
2. *$\Gamma \vdash \mathsf{CanRead}(p_i, \ell_i) \mathbin{@} g \cdot p_{i-1}\langle \ell_i \rangle$ and*
   $\Gamma \vdash \mathsf{CanWrite}(p_{i-1}, \ell_i) \mathbin{@} g \cdot p_i\langle \ell_i \rangle$.

Then there exists some sequence $(q_0, \ell'_0), \ldots, (q_n, \ell'_n)$ where $n \leq m$ such that $q_0 = p_0$, $\ell'_0 = \ell_0$, $q_n = p_m$, and $\Gamma \vdash \ell'_n \sqsubseteq \ell_m \mathbin{@} g \cdot p_m\langle \ell_m \rangle$, and for all $i \in [1, n]$,

- $\Gamma \vdash \ell'_{i-1} \sqsubseteq \ell'_i \mathbin{@} g \cdot q_{i-1}\langle \ell'_i \rangle$,
- $\Gamma \vdash \mathsf{CanRead}(q_i, \ell'_i) \mathbin{@} g \cdot q_{i-1}\langle \ell'_i \rangle$,
- $\Gamma \vdash \mathsf{CanWrite}(q_{i-1}, \ell'_i) \mathbin{@} g \cdot q_i\langle \ell'_i \rangle$,
- *$i = n$ or* $\Gamma \vdash \mathsf{CanWrite}(q_i, \ell'_i) \mathbin{@} g \cdot q_{i-1}\langle \ell'_i \rangle$,

Proof. This is a proof by induction on m.

If $m = 0$ then $n = 0$ and $(q_0, \ell'_0) = (p_0, \ell_0)$ and the case is complete.

If $m = 1$, we consider the two cases of the hypothesis. If $p_0 = p_1$, then we let $n = 0$, set $(q_0, \ell'_0) = (p_0, \ell_0)$. Since $\Gamma \vdash \ell_0 \sqsubseteq \ell_1 \mathbin{@} g \cdot p_0\langle \ell_1 \rangle$, this completes the case. If $p_0 \neq p_1$, then we let $n = 1$ and set $(q_i, \ell'_i) = (p_i, \ell_i)$. This means that $\Gamma \vdash \ell'_1 \sqsubseteq \ell_1 \mathbin{@} g \cdot p_1\langle \ell_1 \rangle$ by FlowsToRefl, and all other conditions are satisfied by assumptions.

We now assume $m \geq 2$ and, inductively, that the lemma is true for all shorter sequences. We first examine the relationship between (p_0, ℓ_0) and (p_1, ℓ_1).

If $p_0 = p_1$, then we have both

$$\Gamma \vdash \ell_0 \sqsubseteq \ell_1 \mathbin{@} g \cdot p_0\langle \ell_1 \rangle$$

$$\Gamma \vdash \ell_1 \sqsubseteq \ell_2 \mathbin{@} g \cdot p_0\langle \ell_2 \rangle$$

Using VarR on the first wit the second as the side condition, and then FlowsToTrans, we can prove $\Gamma \vdash \ell_0 \sqsubseteq \ell_2 \mathbin{@} g \cdot p_0\langle \ell_2 \rangle$. Moreover, because $p_0 = p_1$, any relationships between p_1 and (p_2, ℓ_2) obviously hold between p_0 and (p_2, ℓ_2). Therefore the shorter sequence $(p_0, \ell_0), (p_2, \ell_2), \ldots, (p_m, \ell_m)$ satisfies the premises of our inductive hypothesis, and an inductive application solves this case.

We now examine the case where both permissions hold — i.e., where both $\Gamma \vdash \mathsf{CanRead}(p_1, \ell_1) \mathbin{@} g \cdot p_0\langle \ell_1 \rangle$ and $\Gamma \vdash \mathsf{CanWrite}(p_0, \ell_1) \mathbin{@} g \cdot p_1\langle \ell_1 \rangle$. Since

- p_0 is wise at ℓ_0,
- $\Gamma \vdash \ell_0 \sqsubseteq \ell_1 \mathbin{@} g \cdot p_0\langle \ell_0 \rangle$, and
- $\Gamma \vdash \mathsf{CanRead}(p_1, \ell_1) \mathbin{@} g \cdot p_0\langle \ell_1 \rangle$,

we know that p_1 is trustworthy with respect to p_0 at ℓ_1 and p_1 is wise at ℓ_1. We can therefore apply our inductive hypothesis to $(p_1, \ell_1), \ldots, (p_m, \ell_m)$, producing some $(q_1, \ell'_1), \ldots, (q_n, \ell'_n)$ where $q_1 = p_1$, $\ell'_1 = \ell_1$, and $n \leq m$. The trustworthiness of p_1 with respect to p_0 at ℓ_1 gives us two sub-cases to consider.

If $\Gamma \vdash \mathsf{CanWrite}(p_1, \ell_1) \mathbin{@} g \cdot p_0\langle \ell_1 \rangle$, then we can simply add (p_0, ℓ_0) to the front of the sequence and satisfy all necessary conditions, proving the case.

The last sub-case is slightly more involved. Since $\Gamma \vdash \ell_1 \sqsubseteq \ell'_2 \mathbin{@} g \cdot p_1\langle \ell'_2 \rangle$, p_0 believes the same thing at ℓ'_2. Using the assumption that $\Gamma \vdash \ell_0 \sqsubseteq \ell_1 \mathbin{@} g \cdot p_0\langle \ell_1 \rangle$, VarR, and FlowsToTrans, we can therefore prove

$$\Gamma \vdash \ell_0 \sqsubseteq \ell'_2 \mathbin{@} g \cdot p_0\langle \ell'_2 \rangle.$$

By the second condition on trustworthiness, and $\Gamma \vdash \mathsf{CanRead}(q_2, \ell_2') \mathbin{@} g \cdot p_1\langle \ell_2' \rangle$, we also know that

$$\Gamma \vdash \mathsf{CanRead}(q_2, \ell_2') \mathbin{@} g \cdot p_0\langle \ell_2' \rangle.$$

Next we claim that $\Gamma \vdash \mathsf{CanWrite}(p_0, \ell_2') \mathbin{@} g \cdot q_2\langle \ell_2' \rangle$. We already know that we have proofs of the following:

$$\Gamma \vdash \mathsf{CanRead}(p_2, \ell_2') \mathbin{@} g \cdot p_1\langle \ell_2' \rangle$$

$$\Gamma \vdash \mathsf{CanWrite}(p_1, \ell_2') \mathbin{@} g \cdot q_2\langle \ell_2' \rangle$$

$$\Gamma \vdash \mathsf{CanWrite}(p_0, \ell_1) \mathbin{@} g \cdot p_1\langle \ell_1 \rangle$$

$$\Gamma \vdash \ell_1 \sqsubseteq \ell_2' \mathbin{@} g \cdot p_1\langle \ell_2' \rangle$$

Applying the rules VarR then CWVar to $\Gamma \vdash \mathsf{CanWrite}(p_0, \ell_1) \mathbin{@} g \cdot p_1\langle \ell_1 \rangle$ using the flow relationship as the side condition both times yields a proof of the sequent $\Gamma \vdash \mathsf{CanWrite}(p_0, \ell_2') \mathbin{@} g \cdot p_1\langle \ell_2' \rangle$. We can now forward this belief to q_2, using FwdR and the first two beliefs above as side conditions, to produce a proof of

$$\Gamma \vdash \mathsf{CanWrite}(p_0, \ell_2') \mathbin{@} g \cdot q_2\langle \ell_2' \rangle.$$

We have now proven all required conditions between p_0 and (q_2, ℓ_2') except the final bullet. We do not do this directly, but rather note that the first three conditions are sufficient to satisfy the *premises* of the lemma, and thus our inductive hypothesis. Since $n \leq m$, we therefore have that $(p_0, \ell_0), (q_2, \ell_2'), \ldots, (q_n, \ell_n')$ satisfies the conditions of our inductive hypothesis. The output of this application proves the case. □

We additionally prove one more simple lemma.

Lemma 7. *If $\Gamma \vdash g \cdot p\langle \ell \rangle \ \mathsf{SF}\ g'$ is provable without using* ExtSF, SelfLSF, *or* SelfRSF, *then g' is of the form $g \cdot q\langle \ell' \rangle$.*

Proof. This argument follows by induction on the proof of $\Gamma \vdash g \cdot p\langle\ell\rangle \ \mathsf{SF} \ g'$. The cases for ReflSF, VarSF, and FwdSF are trivial, as they can only prove speaks-for relationships of the desired form.

For TransSF, we have

$$\frac{\Gamma \vdash g \cdot p\langle\ell\rangle \ \mathsf{SF} \ g'' \qquad \Gamma \vdash g'' \ \mathsf{SF} \ g'}{\Gamma \vdash g \cdot p\langle\ell\rangle \ \mathsf{SF} \ g'}$$

By induction on the first hypothesis, we have that $g'' = g \cdot p'\langle\ell''\rangle$ for some p' and ℓ''. Therefore the right premise becomes $\Gamma \vdash g \cdot p'\langle\ell''\rangle \ \mathsf{SF} \ g'$. Applying the inductive hypothesis again to this premise proves that $g' = g \cdot q\langle\ell'\rangle$ for some q and ℓ', thus proving the case. □

Theorem 20 (Respect of Permission Beliefs)**.** For all contexts Γ, generalized principals g, regular principals p and q, and labels ℓ and ℓ', if $\Gamma \vdash g \cdot p\langle\ell\rangle \ \mathsf{SF} \ g \cdot q\langle\ell'\rangle$ is provable without using ExtSF, SelfLSF, or SelfRSF and p is wise at ℓ in world (g, Γ), then there exists a sequence $(p_0, \ell_0), \ldots, (p_n, \ell_n)$ such that $p_0 = p$, $\ell_0 = \ell$, $p_n = q$, $\Gamma \vdash \ell_n \sqsubseteq \ell' \ @ \ g \cdot q\langle\ell'\rangle$, and for all $i \in [1, n]$,

- $\Gamma \vdash \ell_{i-1} \sqsubseteq \ell_i \ @ \ g \cdot p_{i-1}\langle\ell_i\rangle$,
- $\Gamma \vdash \mathsf{CanRead}(p_i, \ell_i) \ @ \ g \cdot p_{i-1}\langle\ell_i\rangle$,
- $\Gamma \vdash \mathsf{CanWrite}(p_{i-1}, \ell_i) \ @ \ g \cdot p_i\langle\ell_i\rangle$,
- $i = n$ or $\Gamma \vdash \mathsf{CanWrite}(p_i, \ell_i) \ @ \ g \cdot p_{i-1}\langle\ell_i\rangle$.

Proof. This proof proceeds by first constructing a sequence of principal-label pairs satisfying the hypothesis of Lemma 6, and then applying the lemma.

To construct such a sequence, we induct on the proof of $\Gamma \vdash g \cdot p\langle\ell\rangle \ \mathsf{SF} \ g \cdot q\langle\ell'\rangle$. We consider each rule that may occur in that proof separately. For ReflSF, we

know $(p, \ell) = (q, \ell')$, so we simply output $(p_0, \ell_0) = (p, \ell)$. For VarSF, we know that $p = q$ and $\Gamma \vdash \ell \sqsubseteq \ell' \mathbin{@} g \cdot p\langle \ell' \rangle$, so we output $(p, \ell), (p, \ell')$. For FwdSF, we know that $\ell = \ell'$, so we output $(p, \ell), (q, \ell)$, and the conditions on FwdSF are exactly those needed when $p \neq q$.

For the TransSF case, Lemma 7 tells us we have a proof of the form

$$\frac{\Gamma \vdash g \cdot p\langle \ell \rangle \ \mathsf{SF} \ g \cdot p'\langle \ell'' \rangle \qquad \Gamma \vdash g \cdot p'\langle \ell'' \rangle \ \mathsf{SF} \ g \cdot q\langle \ell' \rangle}{\Gamma \vdash g \cdot p\langle \ell \rangle \ \mathsf{SF} \ g \cdot q\langle \ell' \rangle}$$

In this case, we acquire sequences $(p_0, \ell_0), \ldots, (p_a, \ell_a)$ and $(q_0, \ell'_0), \ldots, (q_b, \ell'_b)$ by inductively applying the procedure to both premises. We note that $(p_0, \ell_0) = (p, \ell)$ and $(q_b, \ell'_b) = (q, \ell')$, but also $(p_a, \ell_a) = (q_0, \ell'_0)$. This means we can simply connect the two sequences, producing the following sequence with all necessary properties

$$(p_0, \ell_0), \ldots, (p_a, \ell_a), (q_1, \ell'_1), \ldots, (q_b, \ell'_b).$$

This sequence, by construction, satisfies the premises of Lemma 6. By simply applying the lemma to this sequence, we thus prove the theorem. □

CHAPTER 5

FUTURE WORK

In this chapter, we discuss future extensions of the work in the preceding chapters, divided into three parts. First, in Section 5.1, we consider extensions to the work in Chapter 2 on effectful programs. Then, in Section 5.2, we consider extensions to the work in Chapters 3 and 4 on authorization logic. Finally, in Section 5.3, we consider bringing these two lines of research together.

5.1 Semantics of Effectful Programs

In Chapter 2, we looked at a technique for combining monadic and comonadic effects using layering, giving a denotational account of strictness and laziness in the process. We also noted that layering can be used to give a lazy account of other producer effects by layering a monad with the ! comonad. In this section, I propose three projects designed to expand the theory of layering.

5.1.1 Probabilistic Game Semantics are Monadic

As fields like machine learning and differential privacy become more popular, probabilistic programming is ascendant. Probabilistic programming languages provide primitives for creating probability distributions over data types and for manipulating those distributions, such as by creating conditional distributions. Higher-order probabilistic languages thus need to define and manipulate distributions over higher-order types, including function types.

Usually we give semantics to a probabilistic programming language using a

Giry monad [Giry, 1982], as originally suggested by Kozen [1981]. However, the category of probability spaces is not Cartesian-closed [Jung and Tix, 1998], and so this semantics does not scale to higher-order languages. In particular, probability distributions are defined via a construction called a σ-algebra, and the space of structure-preserving functions between σ-algebras is not a σ-algebra. This has inspired a long line of work attempting to generalize σ-algebras to spaces appropriate for higher-order probabilistic programming, such as quasi-Borel predomains [Vákár et al., 2019].

Another line of work abandons monadic semantics for probabilistic programs, preferring *game semantics*. Game semantics views programs as strategies for interacting with the environment, or proofs as strategies for interacting with an opponent of a proposition. Originally, game semantics were invented for linear logic by Blass [1992]. Intuitively, states of play in Blass's semantics are "used up" every round, modeling linearity. Later, game semantics was updated for programming languages by Hyland and Ong [1993] and independently by Abramsky, Jagadeesan, and Pasquale [1994]. This semantics operates under the ! comonad of linear logic, allowing them to give semantics to non-linear languages. Still later, game semantics was adapted for probabilistic programming by Danos and Harmer [2002] and Winskel [2013, 2015].

A deeper dive reveals that the work on probabilistic game semantics gives semantics exclusively to lazy languages. From Chapter 2, we know that the Kleisli category of the ! comonad represents lazy programs, so this is not surprising. However, this observation immediately leads to another: any Giry-like monad implicit in probabilistic game semantics would appear on both the input and the output, rather than just on the output.

With this observation, it is easy to see how we might show that probabilistic game semantics are, in fact, monadic. First, we must develop a *linear* game semantics with appropriate structure to define both a Giry-like monad and a ! comonad. Melliès and Tabareau [2007] give a recipe to develop the ! comonad. In order to develop the Giry-like monad, the move set of these games must be σ-algebras. Then, the Giry-like monad sends a game G to a game where the moves are the probability distributions of the moves of G.

Once these are developed, Danos and Harmer's [2002] games should correspond to the comonad-prioritizing layering, while the monad-prioritizing layering corresponds to a probabilistic game semantics made strict in the style of Abramsky and McCusker [1997] and Honda and Yoshida [1997].

However, there are still design challenges. For instance, strategies can be viewed as partial functions that take a list of moves and produce a move, representing the ability of a player to choose their move based on the current state of play. However, the above description of a Giry-like monad for game semantics would then view strategies as partial functions from lists of *distributions over moves* to a distribution over moves. This would represent allowing a program to see the probability distributions previous states were drawn from, but not the states themselves.

If we can overcome these challenges, we would have a monadic semantics of higher-order probabilistic programming that is much simpler than quasi-Borel predomains. A monadic semantics for higher-order probabilistic programming would allow us to match the informal semantics of several probabilistic programming languages.

5.1.2 Strict and Lazy Semantics for Effect Systems

The theory of effects originally stems from *effect systems* [Lucassen and Gifford, 1988; Marino and Milstein, 2009; Nielson, 1996; Nielson and Nielson, 1999], extensions of type systems used to analyze code for producer effects. When Wadler and Thiemann [1998] argued for the "marriage of effects and monads," they focused on how effect systems can be given semantics via *generalizations* of monads designed to allow more than one effect. A long line of later work has explored this idea [Atkey, 2009; Filinski, 1999; Katsumata, 2014; Tate, 2013].

More recently, there has been a surge of work focused on analyses for systems of consumer effects, called *coeffect systems* [Brunel et al., 2014; Petricek et al., 2012, 2014]. These are mostly used for analyzing resource usage by splitting the linear modality ! into many effects $!_n$, representing a resource that must be used exactly n times.

So far, little ink has been spilled on the semantics of *effect-and-coeffect systems*, which analyze code for both producer effects and consumer effects. In fact, the only previous work on such semantics uses distributive laws [Gaboardi et al., 2016]. Since the doubly-effectful languages of Chapter 2 can be viewed as limited effect-and-coeffect systems, we have already shown that distributive laws do not always exist for effect-and-coeffect systems. We would like to expand the theory of layering to other effect-and-coeffect systems. This would not only be a theoretical win, but would allow us to expand the resource usage analyses described above to lazy languages.

5.1.3 Connections with Adjunction Models

As we pointed out in Chapter 2, strictness and laziness are extremely important phenomena in programming languages which have been with us since the beginning [Church and Rosser, 1936]. Moreover, many algorithms and data structures vitally use laziness for their time and space complexities [Okasaki, 1996]. Haskell programs, among others, commonly use laziness to represent infinite data structures [Friedman and Wise, 1976].

Despite the importance of laziness, the denotational semantics of laziness is a shockingly modern topic. For instance, in domain theory—the premier style of denotational semantics—one can express *non-strictness* much more easily than one can express true laziness. In the past two decades there have been several attempts at denotational models of laziness. For instance, Zeilberger [2009] uses *polarization*, an idea from logic, to give an account of strictness and laziness.

Several models of laziness come from denotational semantics for calculi with mixed strictness and laziness. The most successful of these calculi is Levy's Call-By-Push-Value [Levy, 1999, 2001]. Another equally-interesting line of work is the $\lambda\mu$-calculus of Parigot [1992a,b] and the $\bar{\lambda}\mu\tilde{\mu}$-calculus of Curien, Fiore, and Munch-Maccagnoni [2016], which are designed based on sequent calculus for classical logic.

Interestingly, all of these semantics are based on categorical structures called *adjunctions*, which are deeply connected to monads and comonads. Importantly, every adjunction gives rise to a monad and a comonad through composition. Even more importantly, every monad (and every comonad) gives rise to many adjunctions, and the Kleisli category is the *initial* such adjunction. Perhaps this means that layering decomposes into an adjunction model in a way that gives deep con-

nections to previous models. However, adjunction models use (mostly) arbitrary adjunctions, while layering uses specific properties of Kleisli categories. Overcoming this difficulty will expose deep connections between effects and Call-By-Push-Value calculi, $\lambda\mu$-calculus, and $\bar{\lambda}\mu\tilde{\mu}$-calculus. Moreover, we may be able to develop effect-and-coeffect systems for languages with mixed strictness and laziness.

5.2 Semantics of Authorization Policies

In Chapter 3, we looked at belief models of authorization logic, and in Chapter 4 we looked at a first-order logic for flow-limited authorization. In this section, we propose several projects extending the semantics of authorization policies, especially flow-limited authorization.

5.2.1 Model Theory for FLAFOL

In Chapter 3, we discussed how a soundness theorem for a proof system can assure correctness of the proof rules. However, in Chapter 4 we presented FLAFOL without providing any model-theoretic semantics. In this section, we propose a belief semantics for FLAFOL.

Developing belief semantics for FLAFOL poses some serious challenges. In particular, FLAFOL uses formal beliefs and generalized principals rather than Says-ILR to give rules for manipulating modalities, unlike FOCAL. Moreover, FLAFOL's interpretation of implications (see Section 4.4.6) does not have any obvious counterpart in belief semantics.

Instead, we propose a belief semantics for a new logic, Limited FLAFOL, which combines the designs of FLAFOL and FOCAL. Like FOCAL, Limited FLAFOL would not use formal beliefs, but would instead use (a version of) the Says-ILR, Says-IL, and Says-IR rules. This means that the Fwd rule would work only on statements of the form p says_ℓ φ. However, it not only connects to FOCAL and belief semantics, but also has significant proof-theoretic advantages, since says now distributes over implication (see Section 4.4.6), but not disjunction (see Section 4.3).

5.2.2 Secure Checking of FLAFOL proofs

Recall the example from Section 4.1.1 in which Alice attempts to view Bob's photo on social media. In this example, Bob may set his policy such that only those on his friend list may view his photo. In order to view Bob's photo, Alice must prove to the system that she is on Bob's friend list, which may require communication with Bob.

Now, imagine that Alice runs the following program:

```
if (secret) {
  look at Bob's picture
} else {
  do nothing
}
```

If every principal knows the program text (a standard assumption in information flow), then Bob knows Alice's `secret` variable is `true` if Alice communicates with him to prove that she is his friend. If Bob is not supposed to know this secret,

Alice's information has leaked. Sadly, since FLAFOL knows nothing about `secret` its non-interference theorem fails to protect against this leak.

This particular type of leak is called an *implicit flow*. Standard information-flow-control type systems combat implicit flows using *PC labels*, which represent the labels of any information used to get to a program point. Importing this idea to FLAFOL would limit participation in a proof to those who may know and influence the current context. Thus the above program would not be allowed to compile compile unless Bob is allowed to know the value of `secret`.

This logic has a simple design, though with some surprises. For example, in order to eliminate cuts every formula must have its own PC label since proofs cut in for assumptions may themselves involve communication. While the design is simple, determining an appropriate guarantee is harder. The guarantee should capture the intuition that secrets in the environment do not leak. One possible guarantee axiomatizes the information principals learn when checking a proof.

Pursuing this strategy would mean that we need an extra security guarantee on the cut-elimination procedure since we need to know whether cut elimination allows principals to learn more information during proof checking. This guarantee would allow system designers to use cut freely when proving meta-theorems, without fear that eliminating those cuts will weaken their guarantees.

5.2.3 Temporal Authorization Logic

Imagine that Alice and Bob are trying to engage in an atomic swap, where Alice and Bob want to exchange items without either of them ever having access to both. They can implement this by having box A and box B, where Alice has

access to box A and Bob has access to box B. When they are ready to swap, both Alice and Bob place their items in their respective boxes. Whenever both boxes are filled, Alice loses access to box A and gains access to box B, while Bob loses access to box B and gains access to box A. This is clearly an authorization policy. Unfortunately, since authorization logics do not have a notion of time built in, no authorization logic (including FOCAL and FLAFOL) can reason about this policy.

Other logics do attempt to reason about programs over time. Temporal logics [Pnueli, 1977] divide time into logical steps by providing modalities representing how formulae can vary over time. For example, temporal logics usually include:

- a modality X with $X\varphi$ representing "φ holds in the next step."
- a modality $\Box$ with $\Box\varphi$ representing "φ holds from now on."
- a connective $\mathcal{U}$ with $\varphi \,\mathcal{U}\, \psi$ representing "ψ holds at some point in the future, and φ holds until that time."

No previous work of which we are aware has combined authorization logic with temporal logic. Such a logic would be able to reason about dynamic policies. Previous work [Garg and Pfenning, 2010] focused on authorization logics for *time-limited* policies which can be tagged with a statement like expires at 5PM. However, time-limited policies cannot describe atomic swap as described above.

The non-interference statement we used for FLAFOL (Theorem 19) does not work if authorization may by revoked, which would naturally show up in temporal authorization logic. An appropriate analogue for the temporal setting must be found. Moreover, models of authorization logic (which focus on the semantics of says) and models of temporal logic (which focus on time) are difficult to combine.

5.3 Combining Semantics for Programs and Policies

So far we have argued that one must understand the semantics of both effectful programs and security policies in order to create proven-secure programs. However, we have yet to discuss the highly-nontrivial work required to bring these lines of research together. It is critically important that we do bring them together, since in order to verify that a security mechanism does indeed enforce a security policy, we need to show that the semantics of the mechanism match the semantics of the policy. In this section, we propose three projects on this interface.

5.3.1 Producer Effects in Information Flow

In Section 5.2.2, we discussed how standard information-flow-typed languages use PC labels to prevent implicit flows by collecting information on the labels used in the control flow. They then require that any principal who can see an effect be able to read the PC, and also that any principal who can change the effect be able to write the PC.

Recall that monads capture effects by changing the output type of a program, operationalizing the effect and rendering that program pure. When a monad is defined on a language with information-flow labels in the types, the monad can also change the labels in that output type. This represents both which principals may see the effect and which principals may control the effect. For instance, the state monad in a language with information flow types would be

$$\mathsf{State}(X\{\ell\}) = \sigma\{\ell_\sigma\} \to (X\{\ell\} \times \sigma\{\ell_\sigma\})\{\bot\},$$

where ℓ_σ represents the label on the state. This represents programs with a state

which can be both read from and written to, but which only principals that can read ℓ_σ can see, and only principals that can write ℓ_σ can affect.

This state monad with information-flow types might look a little different than expected. In previous work [Crary et al., 2005; Devriese and Piessens, 2011; Harrison and Hook, 2005; Jia and Zdancewic, 2009; Li and Zdancewic, 2006; Russo et al., 2008; Stefan et al., 2011; Tsai et al., 2007], information flow and monadic effects have been combined very differently. Instead of focusing on effects in language with information-flow security types, these papers look at enforcing information flow using monads, a technique pioneered by Abadi et al. [1999]. Combining this method with the monad design pattern for functional languages allowed these authors to bake reasoning about effects into their information-flow enforcement.

Let us look at what that reasoning about effects in information-flow enforcement looks like: imagine that we have a program such as

```
if Alice.secret then write(3) else write(4),
```

which represents a branch on a secret of Alice's, causing either 3 or 4 to be written to the state. This program will only type-check if Alice may control the data written to the state and Alice allows her secrets to be written to the state. In order to enforce this, in each branch the PC is tainted with Alice's confidentiality and integrity, which ensures that the `write` commands cannot type check otherwise.

Similar reasoning, changing only the label, works for many effects. We would like to understand this reasoning as generic over effects, assuming only that we have a system of producer effects. Concretely, we would like to show that if we translate this into a pure monadic program, then the normal check on if statements subsumes this PC check. In order to do this we will need to create a language with information-flow types which can express monads representing the most common

effects in the information-flow-control literature. We can then develop monadic translations from an effectful language, and show that the PC checks in this effectful language correspond to monadic operations. In particular, we would like to consider non-termination as an effect, deriving the rules for termination-sensitive information-flow control; and to consider state as an effect, deriving the PC checks required for state.

Once we have this, we can apply layering to get a semantics for a lazy language with information-flow types. Such languages have never been formally considered before, though explorations of information flow often take place in Haskell [Li and Zdancewic, 2006; Russo et al., 2008; Stefan et al., 2011; Tsai et al., 2007; Waye et al., 2015]. Since PCs correspond to labels on effects (such as ℓ_σ above) layering suggests that you should need a PC on each input. We also see this in FLAFOL due to cut elimination (see Section 5.2.2). This further suggests a method for developing information-flow type systems for languages with full β-reduction, though that is mostly of theoretical interest.

5.3.2 Distributed Modal Type Theory

It is always interesting, and often enlightening, to think about a propositions-as-types interpretation of a new logic. In particular, we have flirted with propositions-as-types interpretations of authorization logic several times in this dissertation. Such an interpretation would be a program calculus with a notion of location, a notion of communication between locations, and a notion of trust.

Previous work by Ahmed, Jia, and Walker [2003], Jia and Walker [2004], and Murphy [2008] used modal logic as a basis for distributed programming. These used

hybrid logic which allows formulae to reference Kripke-style possible worlds. Under this interpretation, each computer in the system is a world of a Kripke model. However, they then have difficulties defining β-reduction for the same reason that we had difficulty proving cut in a version of FLAFOL where says distributes across implication (see Section 4.4.6). Thus, Murphy [2008] develops a notion of *mobile code* in order to disallow communication of functions that reference local data.

Authorization logics reason about distributed programs from the top down, considering the actions of all principals at once. Choreographies [Montesi, 2013] provide a way of writing distributed programs from the top down, directing all participants at once, and compiling these programs to code for each individual process. We propose formalizing this into a propositions-as-types connections, though this is a non-trivial task since choreographies are usually presented untyped. There is some work on types for choreographies [Carbone and Montesi, 2013; Carbone et al., 2014; Cruze-Filipe and Montesi, 2017], but this work uses session types. Session types constrain the behavior of a program to follow some protocol. The program already specifies the protocol, so this simply repeats code, which is an unsatisfying role for a type system.

5.3.3 A Distributed, Modal, Dependently-Typed Language

Let us now consider a longer-term and thus more-speculative project. With current proof assistants, writing machine-checked proven-secure distributed programs is incredibly difficult. There are essentially two choices: (a) write each component separately and prove things about the component without any guarantees about the system as a whole, or (b) create a virtual machine and prove things about the virtual machine instead, a massive effort which obliterates many of the advantages

of proof assistants. There is no language on the market for writing proofs about distributed programs. We propose further extending the ideas from Section 5.3.2 to provide such a language.

We can think of dependently-typed languages as *polymorphic* languages which have forgotten the difference between types and programs. [Jacobs, 1999] suggests starting from a logic for programs—that is, a second-order logic where terms include programs and reasoning about computation—and bootstrapping from there. While this has worked well in the case of intuitionistic logic, it has proven to be much more difficult to make work for more-complicated logics, like linear logic McBride [2016]. Dependent modal type theory has been considered before [Birkedal et al., 2019; de Paiva and Ritte, 2016; Nanevski et al., 2007], but never in the context of distributed computing. There is significant theoretical and practical challenge in building such a type theory.

However, we would like to do so with a language built on an authorization logic like FLAFOL or FOCAL. Using the reasoning from Section 5.3.2, we can see that this bootstrapped language would provide choreographic distributed programming as well as the full power of dependent types. Moreover, it could encode security constraints using authorization logic and information-flow-control types. This would provide a single framework in which distributed systems could be written, the security of the system expressed, and the system be proven secure.

CHAPTER 6

RELATED WORK

6.1 Effects, Monads, and Comonads

Monads and Effects Monads are the centerpoint of most of the research on effects. This began with Moggi's [1989] seminal paper on the work. He defined a monadic notion of computation, using strong monads to extend the λ-calculus with operations such as throwing exceptions and reading and writing state.

At about the same time, Lucassen and Gifford [1988] were developing the first type-and-effect system. Type-and-effect systems classify programs as having effects from some set, and give ways of combining these effects. Lucassen and Gifford also provided an effect-inference algorithm, analogous to a type-inference algorithm.

Wadler and Thiemann [1998] developed an indexed version of Moggi's monadic semantics for Lucassen and Gifford's type-and-effect system, building a bridge between these areas. Since then, significantly more research has been done in both monadic semantics [Atkey, 2009; Filinski, 1999; Hicks et al., 2014; Hyland et al., 2006, 2007; Lüth and Ghani, 2002; Peyton Jones and Wadler, 1993] and type-and-effect systems [Marino and Milstein, 2009; Nielson, 1996; Nielson and Nielson, 1999]. Many of these later developments required generalizations of the previous work. Tate [2013] unified these generalizations with a formalization of producer effects and a semantics for producer effect systems, which he proved to be as general as possible.

Comonads Comonads and consumer effects, or "coeffects," have not been as thoroughly studied as monads and producer effects. The first use of comonads in computer science comes from Brookes and Geva [1992]. They gave a denotational account of how computations use inputs. In particular, their semantics showed how different computations with different evaluation orders might have different meanings. They discovered that by giving semantics in domain theory with comonadic computations, one can distinguish between amounts of computation done on different inputs.

Several years later, Uustalu and Vene [2008] identified many more applications of comonads, and showed them to be difficult to express monadically. A particularly interesting example was their use of comonadic computation to give meaning to dataflow languages [Uustalu and Vene, 2005]. Uustalu and Vene's work considered a single comonad at a time, whereas Petricek, Orchard, and Mycroft [2012, 2014] identified a way to index comonads. Brunel, Gaboardi, Mezza, and Zdancewic [2014] then adapted Petricek et al.'s work to comonads with weakening and contraction.

Combining Monads and Comonads We are not the first to discover the Kleisli-like constructions in Section 2.4, although we did strengthen their connection to effects. Brookes and Geva [1992] had discovered three comonads similar to ! for domain theory. Brookes and van Stone [1993] later investigated how to combine these comonads with various monads in a general fashion in order to connect with Moggi's monadic notions of computation. In doing so, they developed the constructions of Section 2.4. However, they focused on distributive laws because they did not identify the equivalence in Theorem 6 that implies that distributive laws cannot be used for the application they were striving for. On the

other hand, Power and Watanabe [2002] developed a 2-categorical treatment of the constructions, even identifying the equivalence in Theorem 6. However, Power and Watanabe did not identify applications to any particular monad and comonad without a distributive law. Therefore, we are the first to recognize that there are important applications in the theory of effects where a distributive law cannot exist, such as strictness versus laziness.

We are the first we are aware of to formalize a notion of doubly-effectful languages. This means that we are the first to prove that the semantics using distributive laws is complete as well as sound for the subsumption laws we presented for $K^{\sigma}_{C,M}$. We are also the first to note that in some doubly-effectful languages, producer-effectful programs or consumer-effectful programs may not be able to embed into the doubly-effectful language.

More recently, Gaboardi, Katsumata, Orchard, Breuvart, and Uustalu [2016] have extended the theory of distributive laws to cover graded monads and comonads. As an example, they apply this to show that information flow commutes with non-determinism. That is, they prove that following a non-deterministic program with a program whose access to confidential data is limited does not unintentionally leak more data to that program.

6.2 Strictness and Laziness

The divide between strict and lazy evaluation strategies has long been a point of interest [Ariola et al., 1995; Levy, 1999, 2001; Maraist et al., 1995; Plotkin, 1975; Sabry and Wadler, 1997; Zeilberger, 2009], even since the beginning of computer science [Church and Rosser, 1936]. Plotkin [1975] described the two main ways of

providing languages with strict and lazy evaluation orders: call-by-value and call-by-name. Decades later, this led to a series of papers that gave several evaluation strategies with varying properties [Ariola et al., 1995; Levy, 1999, 2001; Maraist et al., 1995; Sabry and Wadler, 1997; Zeilberger, 2009].

One of these, by Maraist et al. [1995], used *intutionistic* linear type theory, which among other things does not have the ? exponential, to give semantics to strictness and laziness. More specifically, they in a sense give semantics to different evaluation strategies via the linear λ-calculus invented by Wadler [1990]. They are interested in intensional properties of these translations, such as whether various terms are syntactically identical, whereas we are interested in extensional properties, such as whether various terms interact similarly. Thus, they focus on where to place $\mathsf{supply}_{!x}\{-\}$ to force or delay evaluation, whereas we focus on how to compose programs so that they exhibit specific interactions.

6.2.1 Polarization and Focusing

Other works connect the proof theory of classical logic with strictness and laziness through linear logic. For instance, Danos, Joinet, and Schellinx [1997] develop two new logics, LKT and LKQ, each of which restricts classical logic such that cut elimination is confluent. LKT and LKQ can be viewed through the lens of *polarization*, which was developed by Girard to constructivize classical logic [Girard, 1991]. Polarization associates each basic proposition and connective with a positive or negative *polarity*, and recursively associates each formula with a polarity. Viewed through this lens, LKT corresponds to classical logic with an always-negative interpretation, while LKQ corresponds to classical logic with an always-positive interpretation. They connected LKT and LKQ to Schellinx's [1994]

linear decorating, showing that LKT can be translated into linear logic using a comonad-prioritizing layering, and that LKQ can be translated into linear logic using a monad-prioritizing layering.

An important theorem of Danos et al. [1997] is that Parigot's logic, Free Deduction [Parigot, 1992b], embeds into a unifying logic, LK^{tq}. This allowed them to embed Parigot's $\lambda\mu$-calculus [Parigot, 1992a] into LKT. Parigot developed $\lambda\mu$-calculus by restricting Free Deduction such that η-equalities are respected in cut elimination. Essentially, $\lambda\mu$-calculus gives a method of writing programs with multiple outputs without using parallelism. Instead, it uses names to essentially choose an output to focus on in the program, with mechanisms for switching between named outputs. This gives a computational model of classical logic via a proofs-as-programs correspondence with classical natural deduction.

Our work stands that of Danos et al. [1997] on its head. Instead of developing the layerings via polarization, we directly use the layerings. The layerings force a positive or negative interpretation because of the polarization properties of the linear exponentials. However, by putting the layerings first, we are able to connect to the work on effects through monadic and comonadic semantics.

While Danos et al. did not discuss strictness and laziness, several follow-up pieces of work did. In particular, the $\lambda\mu$-calculus of Parigot [1992a], the $\bar{\lambda}\mu\tilde{\mu}$-calculus of Curien and Herbelin [2000], and the CPS translations of Zeilberger [2009] use polarization to discuss strictness and laziness through Andreoli's 1992 work on *focusing*. Categorical models for $\lambda\mu$-calculus and $\bar{\lambda}\mu\tilde{\mu}$-calculus were explored by Curien, Fiore, and Munch-Maccagnoni [2016]. Interestingly, models for the Effect Calculus of Egger, Møgelberg, and Simpson [2014] and the Call-By-Push-Value model of computation of Levy [2001] are subsumed by Curien et al.'s

models of $\bar{\lambda}\mu\tilde{\mu}$-calculus, which connects that calculus to effects. However, Curien et al. mention that they struggle to develop a calculus with exceptions and handlers that matches their semantics.

6.3 Linear Logic

The exponentials of linear logic are the basis of our motivation for layering. Linear logic was developed by Girard [1987] to give a logic of resources. However, it has been described as "a proof-theorist's logic," since it is often understood as calling out the structural rules of Gentzen's original sequent-calculus formulation of classical logic [Gentzen, 1935a,b].

Linear logic retains Gentzen's cut-elimination theorem, which tells us that the proofs-as-programs construction for classical linear logic is normalizing [Girard, 1987]. In classical logic, there are multiple ways to reduce a cut, but in linear logic there is only one. As a consequence, the proofs-as-programs construction for classical linear logic is also confluent (modulo exchange), which altogether makes it canonicalizing [Bellin and Scott, 1994].

These deep connections between classical logic and classical linear logic led to attempts to embed classical logic into classical linear logic. Girard [1987] was able to embed *intuitionistic* logic into linear logic by prepending every formula on the left of a turnstile with a !. However, he was only able to give an embedding for *cut-free* proofs of classical logic into classcal linear logic by prepending every formula on the left of a turnstile with a ! and every formula on the right of the turnstile with a ?. Later, Girard [1991] used the concept of *polarization* to give a semantics to classical logic through *correlation domains*. Correlation domains are a semantic

object that Girard used to give meaning to classical linear logic.

However, Girard never gave a syntactic translation from classical logic to classical linear logic, though Schellinx [1994] was able to do so in his thesis. He explored several ways of embedding classical-logic proofs into classical linear logic by prepending formulae with exponents, a pattern he called linear decorating. In particular, he then showed two compositional embeddings, which correspond to the two layerings presented in this paper. However, he did not generalize beyond linear logic itself to other monads and comonads or to other languages.

Schellinx noted that his embeddings of classical logic into linear logic were constructivizations. He even connected this to the proofs-as-programs principle. However, he did not explore the operational point-of-view of his embeddings. We have illustrated here using effects that they correspond to strict and a lazy notions of classical reasoning, but others have also illustrated this correspondence by instead using polarization and focusing.

6.4 Authorization Logic

Semantic structures similar to the belief models of Chapter 3 have been investigated in the context of epistemic logic [Eberle, 1974; Fagin et al., 1995; Moore and Hendrix, 1979]. Konolige [Konolige, 1986] proves an equivalence result for classical propositional logic similar to Theorem 8.

Garg and Abadi [2008] give a Kripke semantics for a logic they call ICL, which could be regarded as the propositional fragment of FOCAL. The ICL semantics of says, however, uses *invisible* worlds to permit principals to be oblivious to the

truth of formulas at some worlds. That makes Unit (Section 3.4.1) valid in ICL, whereas Unit is invalid in both FOCAL and FLAFOL.

Garg [2008] studies the proof theory of a logic called DTL_0 and gives a Kripke semantics that uses both invisible worlds and *fallible* worlds, at which false is permitted to be valid. Instead of Unit, it uses the axiom p says $((p \text{ says } \phi) \Rightarrow \phi)$. That axiom is unsound in FOCAL, though it is valid in FLAFOL. DTL_0 does not have a speaksfor connective, nor does it have any analogue of CanRead or CanWrite.

Genovese et al. [2012] study several uses for Kripke semantics with an authorization logic they call $\mathrm{BL}_{\mathrm{sf}}$, which also could be regarded as the propositional fragment of FOCAL. They show how to generate evidence for why an access should be denied, how to find all logical consequences of an authorization policy, and how to determine which additional credentials would allow an access. These questions would also be interesting to address in FOCAL. However, the Kripke semantics of $\mathrm{BL}_{\mathrm{sf}}$ differs from FOCAL's in its interpretation of both says and speaksfor, so the results of Genovese et al. are not immediately applicable to FOCAL.

Garg and Pfenning [2006] present an authorization logic and a non-interference result that ensures untrusted principals cannot influence the truth of statements made by other principals. FLAFOL differs from this logic in two ways. First, FLAFOL supports all first-order connectives while Garg and Pfenning only support implication and universal quantification. Second, Garg and Pfenning only use implications to encode trust, rather than having an explicit trust relation between principals. Abadi [2006] also proves such a property for dependency core calculus (DCC), which is the basis of authorization logic CDD. We believe that similar properties could be proved for FOCAL.

One of the more intriguing consequences of our semantics is that says is not a *monad* [Moggi, 1989] for either FOCAL or FLAFOL. Since Abadi's invention of CDD [Abadi, 2006], says is frequently assumed to satisfy the monad laws, which include Unit. In our semantics, however, Unit is invalid. We don't know whether rejecting the monad laws will have any practical impact on FOCAL or FLAFOL. But the seminal authorization logic, ABLP [Abadi et al., 1993], didn't adopt the monad laws. Likewise, Garg and Pfenning [2010] reject Unit in their authorization logic BL_0; they demonstrate that Unit leads to counterintuitive interpretations of some formulas involving delegation. And Abadi [2003] notes that Unit "should be used with caution (if at all)," suggesting that it be replaced with the weaker axiom $(p \text{ says } \phi) \Rightarrow (q \text{ says } p \text{ says } \phi)$. Genovese et al. [2012] adopt that axiom; in their Kripke semantics, the frame condition that validates it is: $w \leq_p u \leq_q v$ implies $w \leq_q v$. So in rejecting the monad laws, FOCAL and FLAFOL are at least in good company.

6.5 Combining Authorization and Information Security

Prior work in both information flow control and authorization logics has explored connections between authorization and information security. The Decentralized Label Model [Myers and Liskov, 1998] incorporates a notion of ownership into its information flow policies, specifying who may authorize exceptions to a policy.

The Flow-Limited Authorization Model (FLAM) [Arden et al., 2015] was the first information-flow label model to directly consider the confidentiality and integrity of policies when authorizing information flows. Prior work on Rx [Swamy et al., 2006] and RTI [Bandhakavi et al., 2008] enforced information flow policies

via *roles* whose membership are protected with confidentiality and integrity labels.

We deviate from these works in several important ways. First, FLAFOL is a formal authorization logic. Second, we employ both principals and labels, but keep them entirely separate. Many information flow models are defined with respect to an abstract security lattice and omit any direct representation of principals. The Decentralized Label Model [Myers and Liskov, 1998] expresses labels in terms of principals. FLAM [Arden et al., 2015] takes this a step further and represents principals directly as a combination of confidentiality and integrity labels. This view restricts FLAM from reasoning about labels with policies other than confidentiality and integrity, since they might necessitate subtle changes to FLAM's reasoning rules.

Unifying principals and labels also undermines FLAM's effectiveness as an authorization logic. It is convenient to construct complex policies from simpler ones, such as a policy protecting Alice's confidentiality and Bob's integrity. FLAM regards such a compound policy as a principal, but this principal does not represent an actual entity in the system. These principals break the connection between principals and system entities often present in authorization logics. While it is certainly possible to represent these unusual principals in FLAFOL, FLAFOL does not necessarily force a reasoner to break this connection between principals and system entities.

Becker [2012] explores preventing probing attacks, authorization queries which leak secret information, in Datalog-based authorization logics like DKAL [Gurevich and Neeman, 2008] and SecPAL [Becker et al., 2010]. In SecPAL$^+$ [Becker, 2010], Becker proposes a new *can listen to* operator, similar to FLAFOL's CanRead permission, that expresses who is permitted to learn specific statements. However,

can listen to expresses permissions on specific statements, not labels as CanRead does. Moreover, FLAFOL tracks dependencies between statements using these labels, so the security consequences of adding a new permission are more explicit.

The Dependency Core Calculus [Abadi, 2006; Abadi et al., 1999] (DCC) has been used to model both information flow control and authorization, but not at the same time. DCC also has a non-interference property, but like many authorization logics, it employs an external lattice to express trust between principals. FLAFOL supports both finer-grained trust and discoverable trust.

The Flow-Limited Authorization Calculus [Arden and Myers, 2016] uses ideas from FLAM and DCC to support discoverable trust. FLAC and Polymorphic DCC [Abadi, 2006] are based on System F, which contains some elements of second-order logic since it supports universal quantification over types, but does not support some features of first-order logic like existential quantification.

Finally, Aura [Jia and Zdancewic, 2009; Jia et al., 2008] embeds DCC into a language with dependent types in order to explore how authorization logic interacts with programs. Their non-interference result for authorization comes directly from DCC, but they express first-order properties by combining constructs from the programming language with constructs from DCC. This makes it unclear what guarantees the theorem provides. Jia and Zdancewic [Jia and Zdancewic, 2009] encode information-flow labels into Aura as principals and develop a non-interference theorem in the style of information-flow systems [Jia and Zdancewic, 2009]. This setup unfortunately makes it impossible for principals to disagree about the meaning of labels, since the labels themselves define their properties.

CHAPTER 7

CONCLUSION

In Chapter 1 we argued that in order to build machine-checked proven-secure software, we need semantics for both effectful programs and security policies. In the subsequent chapters we presented work that expanded both types of semantics. In Chapter 5 we discussed a vision for future work further expanding both types of semantics, and even bringing the two types together.

We would like to draw attention to the following three themes running throughout this dissertation:

- Modal Logic
- Constructivity
- Propositions as Types

Let us briefly reflect on how each of these themes affect the work presented here and future work using this work as a springboard.

Modal Logic Authorization logics *are* multi-modal logics—a point that we made repeatedly in Chapters 3 and 4—and thus it is easy to see how modal logic is relevant. Moreover, layering is also essentially a modal-logic technique since the ! and ? exponentials of classical-linear logic are modalities.

Constructivity In Chapter 2, we focus on layerings, which correspond to embeddings of classical logic into classical linear logic. Since classical linear logic is constructive, this constructivizes classical logic. Keeping that in mind, it seems

incongruous to note—as we did in Chapter 3—that world experts in authorization logic feel that authorization logic must be constructive in order to preserve evidence. This continued in Chapter 4, where we noted that flow-limited authorization logics ought to reject the law of the excluded middle since it was important that decidability be a meta-property of a collection of policies.

Propositions as Types It is hard to over-emphasize how much the propositions-as-types principle influenced the work in this dissertation. This is most clear in Chapter 2, since we designed Proc to have a propositions-as-types connection with (a fragment of) classical linear logic. However, both FOCAL and FLAFOL take great inspiration from the propositions-as-types point-of-view as well. Because information-flow labels are usually defined as part of the type system of a programming language, we took particular inspiration from propositions-as-types in the design of FLAFOL.

The themes of both modal logic and constructivity can be viewed through the lens of propositions as types. In fact, it is probably no understatement to say that a propositions-as-types perspective is fundamental for understanding the semantics of both effectful programs and security policies, especially as the two come together more and more.

BIBLIOGRAPHY

Martín Abadi. Logic in access control. In *Logic in Computer Science (LICS)*, 2003. doi:10.1109/LICS.2003.1210062.

Martín Abadi. Access control in a core calculus of dependency. In *International Conference on Functional Programming (ICFP)*, 2006. doi:10.1145/1159803.1159839.

Martín Abadi. Variations in access control logic. In *Deontic Logic in Computer Science (DEON)*, 2008. doi:10.1007/978-3-540-70525-3_9.

Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *Transactions on Programming Languages and Systems (TOPLAS)*, 15(4), September 1993. doi:10.1145/155183.155225.

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Principles of Programming Languages (POPL)*, 1999. doi:10.1145/292540.292555.

Samson Abramsky. Proofs as processes. *Theoretical Computer Science (TCS)*, 1994. doi:10.1016/0304-3975(94)00103-0.

Samson Abramsky and Guy McCusker. Call-by-value games. In *Computer Science Logic (CSL)*, 1997. doi:10.1007/BFb0028004.

Samson Abramsky, Radha Jagadeesan, and Malacaria Pasquale. Full abstraction for PCF. In *Theoretical Aspects of Computer Software (TACS)*, 1994. doi:10.1006/inco.2000.2930.

Amal Ahmed, Limin Jia, and David Walker. Reasoning about hierarchical storage. In *Logic in Computer Science (LICS)*, 2003. doi:10.1109/LICS.2003.1210043.

Maximilian Algehed. Short paper: A perspective on the dependency core calculus. In *Programming Languages and Analysis for Security (PLAS)*, 2018. doi:10.1145/3264820.3264823.

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation (JLC)*, 2(3), 1992. doi:10.1093/logcom/2.3.297.

Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *Computer and Communication Security (CCS)*, 1999. doi:10.1145/319709.319718.

Owen Arden and Andrew C. Myers. A calculus for flow-limited authorization. In *Computer Security Foundations (CSF)*, 2016. doi:10.1109/CSF.2016.17.

Owen Arden, Jed Liu, and Andrew C. Myers. Flow-limited authorization. In *Computer Security Foundations (CSF)*, 2015. doi:10.1109/CSF.2015.42.

Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *Principles of Programming Languages (POPL)*, 1995. doi:10.1145/199448.199507.

Aslan Askarov and Andrew C. Myers. Attacker control and impact for confidentiality and integrity. *Logical Methods in Computer Science (LMCS)*, 7(3), September 2011. doi:10.2168/LMCS-7(3:17)2011.

Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming (JFP)*, 19(3–4), July 2009. doi:10.1017/S095679680900728X.

Sruthi Bandhakavi, William Winsborough, and Marianne Winslett. A trust management approach for flexible policy management in security-typed languages. In *Computer Security Foundations (CSF)*, 2008. doi:10.1109/CSF.2008.22.

Lujo Bauer, Scott Garriss, Jonathan M. McCune, Michael K. Reiter, Jason Rouse, and Peter Rutenbar. Device-enabled authorization in the Grey system. In *Information Security Conference (ISC)*, 2005. doi:10.1007/11556992_31.

Moritz Y. Becker. Information flow in credential systems. In *Computer Security Foundations (CSF)*, 2010. doi:10.1109/CSF.2010.19.

Moritz Y Becker. Information flow in trust management systems. *Journal of Computer Security (JCS)*, 20(6), December 2012. doi:10.3233/JCS-2012-0443.

Moritz Y. Becker and Peter Sewell. Cassandra: Distributed access control policies with tunable expressiveness. In *Policies for Distributed Systems and Networks (POLICY)*, 2004. doi:10.1109/POLICY.2004.1309162.

Moritz Y Becker, Cédric Fournet, and Andrew D Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security (JCS)*, 18(4), June 2010. doi:10.3233/JCS-2009-0364.

Emmanuel Beffara. A concurrent model for linear logic. In *Mathematical Foundations of Programming Semantics (MFPS)*, 2005. doi:10.1016/j.entcs.2005.11.055.

Gianluigi Bellin and Philip J. Scott. On the π-calculus and linear logic. *Theoretical Computer Science (TCS)*, 135(1), December 1994. doi:10.1016/0304-3975(94)00104-9.

Jean Bénabou. Catégories avec multiplication. *Comptes Rendus de l'Académie des Sciences Paris*, 258, 1963. URL `http://gallica.bnf.fr/ark:/12148/bpt6k3208j/f1965.image`.

Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. Modal dependent type theory and dependent

right adjoints. *ArXiV*, 2019. URL `https://arxiv.org/abs/1804.05236`. In Submission to Mathematical Structures in Computer Science.

Andreas Blass. A game semantics for linear logic. *Annals of Pure and Applied Logic*, 56(1–3), April 1992. doi:10.1016/0168-0072(92)90073-9.

R. F. Blute, J. R. B. Cockett, and R. A. G. Seely. ! and ?: Storage as tensorial strength. *Mathematical Structures in Computer Science (MSCS)*, 6(4), August 1996. doi:10.1017/S0960129500001055.

Stephen Brookes and Shai Geva. Computational comonads and intensional semantics. In *Applications of Categories in Computer Science*, 1992. doi:10.1017/CBO9780511525902.003.

Stephen Brookes and Katheryn van Stone. Monads and comonads in intensional semantics. Technical Report CMU-CS-93-140, Carnegie Mellon University Department of Computer Science, 1993. URL `https://www.cs.cmu.edu/~brookes/papers/MonadsComonads.pdf`.

Aloïs Brunel, Marco Gaboardi, Damiano Mezza, and Steve Zdancewic. A core quantitative coeffect calculus. In *European Symposium on Programming (ESOP)*, 2014. doi:10.1007/978-3-642-54833-8_19.

Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In *Principles of Programming Languages (POPL)*, 2013. doi:10.1145/2429069.2429101.

Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. In *Concurrency Theory (CONCUR)*, 2014. doi:10.1007/978-3-662-44584-6_5.

Ethan Cecchetti, Andrew C. Myers, and Owen Arden. Nonmalleable information flow control. In *Computer and Communication Security (CCS)*, 2017. doi:10.1145/3133956.3134054.

J. G. Cederquist, Ricardo Corin, M. A. C. Dekker, Sandro Etalle, J. I. den Hartog, and Gabriele Lenzini. Audit-based compliance control. *International Journal of Information Security*, 6(2–3), 2007. doi:10.1007/s10207-007-0017-y.

Peter Chapin, Christian Skalka, and X. Sean Wang. Authorization in trust management: Features and foundations. *ACM Computing Surveys (CSUR)*, 40(3), August 2008. doi:10.1145/1380584.1380587.

Stephen Chong and Andrew C. Myers. End-to-end enforcement of erasure and declassification. In *Computer Security Foundations (CSF)*, 2008. doi:10.1109/CSF.2008.12.

Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3), May 1936. doi:10.2307/1989762.

Andrew Cirillo, Radha Jagadeesan, Corin Pitcher, and James Riely. Do As I SaY! Programmatic access control with explicit identities. In *Computer Security Foundations (CSF)*, 2007. doi:10.1109/CSF.2007.19.

Karl Crary, Aleksey Kliger, and Frank Pfenning. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming (JFP)*, 15 (2), March 2005. doi:10.1017/S0956796804005441.

Luís Cruze-Filipe and Fabrizio Montesi. A core model for choreographic programming. In *Formal Aspects of Component Software (FACS)*, 2017. doi:10.1007/978-3-319-57666-4_3.

Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *International Conference on Functional Programming (ICFP)*, 2000. doi:10.1145/351240.351262.

Pierre-Louis Curien, Marcelo Fiore, and Guillaume Munch-Maccagnoni. A theory of effects and resources: Adjunction models and polarised calculi. In *Principles of Programming Languages (POPL)*, 2016. doi:10.1145/2837614.2837652.

Vincent Danos and Russell S. Harmer. Probabilistic game semantics. *Transactions on Computational Logic (TOCL)*, 3(3), July 2002. doi:10.1145/507382.507385.

Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. A new deconstructive logic: Linear logic. *The Journal of Symbol Logic*, 62(3), September 1997. doi:10.2307/2275572.

Valeria de Paiva and Eike Ritte. Fibrational modal type theory. In *Logic and Semantic Frameworks, with Applications (LSFA)*, 2016. doi:10.1016/j.entcs.2016.06.010.

Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM (CACM)*, 19(5), May 1976. doi:10.1145/360051.360056.

John DeTreville. Binder, a logic-based security language. In *Symposium on Security and Privacy (SSP) (Oakland)*, 2002. doi:10.1109/SECPRI.2002.1004365.

Dominique Devriese and Frank Piessens. Information flow enforcement in monadic libraries. In *Types in Language Design and Implementation (TLDI)*, 2011. doi:10.1145/1929553.1929564.

Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In *Computer Science Logic (CSL)*, 2012. doi:10.4230/LIPIcs.CSL.2012.228.

Rolf A. Eberle. A logic of believing, knowing and inferring. *Synthese*, 26(3–4), April 1974. doi:10.1007/BF00883100.

Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. The enriched effect calculus: Syntax and semantics. *Journal of Logic and Computation (JLC)*, 24 (3), June 2014. doi:10.1093/logcom/exs025.

W. B. Ewald. Intuitionistic tense and modal logic. *Journal of Symbolic Logic*, 51 (1), March 1986. doi:10.2307/2273953.

Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, Cambridge, Massachusetts, 1995. ISBN 9780262061629.

Andrzej Filinski. Representing layered monads. In *Principles of Programming Languages (POPL)*, 1999. doi:10.1145/292540.292557.

Andrzej Filinski. Monads in action. In *Principles of Programming Languages (POPL)*, 2010. doi:10.1145/1706299.1706354.

Gisèle Fischer Servi. Semantics for a class of intuitionistic modal calculi. In *Italian Studies in the Philosophy of Science*. Springer, 1981. doi:10.1007/978-94-009-8937-5_5.

Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. In *European Symposium on Programming (ESOP)*, 2005. doi:10.1007/978-3-540-31987-0_11.

Daniel P. Friedman and David S. Wise. CONS should not evaluate its arguments. In *International Conference on Automata, Languages, and Programming (ICALP)*, 1976. doi:10.1145/130854.130858.

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. Combining effects and coeffects via grading. In *International Conference on Functional Programming (ICFP)*, 2016. doi:10.1145/2951913.2951939.

Deepak Garg. Principal-centric reasoning in constructive authorization logic. In *Intuitionistic Modal Logic and Applications (IMLA)*, 2008.

Deepak Garg and Martín Abadi. A modal deconstruction of access control logics. In *Foundations of Software Science and Computational Structures (FOSSACS)*, 2008. doi:10.1007/978-3-540-78499-9_16.

Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In *Computer Security Foundations Workshop (CSFW)*, 2006. doi:10.1109/CSFW.2006.18.

Deepak Garg and Frank Pfenning. Stateful authorization logic: Proof theory and a case study. In *Security and Trust Management (STM)*, 2010. doi:10.3233/JCS-2012-0456.

Valerio Genovese, Deepak Garg, and Daniele Rispoli. Labeled sequent calculi for access control logics: Countermodels, saturation, and abduction. In *Computer Security Foundations (CSF)*, 2012. doi:10.1109/CSF.2012.11.

Gerhard Gentzen. Untersuchungen über das logische schließen i. *Mathematische Zeitschrift*, 39(1), December 1935a. doi:10.1007/BF01201353.

Gerhard Gentzen. Untersuchungen über das logische schließen ii. *Mathematische Zeitschrift*, 39(1), December 1935b. doi:10.1007/BF01201363.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science (TCS)*, 50(1), 1987. doi:10.1016/0304-3975(87)90045-4.

Jean-Yves Girard. A new constructive logic: Classical logic. Technical Report RR-1443, INRIA, 1991. URL `https://hal.inria.fr/inria-00075117/`.

Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989. ISBN 978-0521371810. doi:10.2307/2274726.

Michèle Giry. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, 1982. doi:10.1007/BFb0092872.

Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Symposium on Security and Privacy (SSP) (Oakland)*, 1982. doi:10.1109/SP.1982.10014.

Yuri Gurevich and Itay Neeman. DKAL: Distributed-knowledge authorization language. In *Computer Security Foundations (CSF)*, 2008. doi:10.1109/CSF.2008.8.

William L. Harrison and James Hook. Achieving information flow security through precise control of effects. In *Computer Security Foundations Workshop (CSFW)*, 2005. doi:10.1109/CSFW.2005.6.

Michael Hicks, Gavin Bierman, Nataliya Guts, Daan Leijen, and Nikhil Swamy. Polymonadic programming. In *Mathematically Structured Functional Programming (MSFP)*, 2014. doi:10.4204/EPTCS.153.7.

Jaakko Hintikka. *Knowledge and Belief*. Cornell University Press, Ithaca, New York, 1962.

Andrew K. Hirsch and Michael R. Clarkson. Belief semantics of authorization logic. In *Computer and Communication Security (CCS)*, 2013a. doi:10.1145/2508859.2516667.

Andrew K. Hirsch and Michael R. Clarkson. Belief semantics of authorization logic coq code, November 2013b.

Andrew K. Hirsch and Ross Tate. Strict and lazy semantics of effects: Layering monads and comonads. *International Conference on Functional Programming (ICFP)*, 2018. doi:10.1145/3236783.

Andrew K. Hirsch, Pedro de Amorim, Ethan Cecchetti, Owen Arden, and Ross Tate. First-order logic for flow-limited authorization: Technical report. Technical report, Cornell, 2019.

Kohei Honda and Nobuko Yoshida. Game theoretic analysis of call-by-value computation. In *International Conference on Automata, Languages, and Programming (ICALP)*, 1997. doi:10.1007/3-540-63165-8_180.

Jon Howell and David Kotz. A formal semantics for SPKI. In *European Symposium on Research in Computer Security (ESORICS)*, 2000. doi:10.1007/10722599_9.

Jonathan Howell. *Naming and Sharing Resources across Administrative Domains*. PhD thesis, Dartmouth College, 2000. URL `https://www.cs.dartmouth.edu/~dfk/jonh/dissertation/thesis-final-single.pdf`.

George Edward Hughes and Max J. Cresswell. *A New Introduction to Modal Logic*. Routledge, 1996. doi:10.4324/9780203028100.

John Martin Elliot Hyland and Luke Ong. Pi-calculus, dialogue games, and PCF. In *Functional Programming Languages and Computer Architecture (FPLCA)*, 1993. doi:10.1145/224164.224189.

Martin Hyland, Gordon Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science (TCS)*, 357(1–3), July 2006. doi:10.1016/j.tcs.2006.03.013.

Martin Hyland, Paul Blain Levy, Gordon Plotkin, and John Power. Combining algebraic effects with continuations. *Theoretical Computer Science (TCS)*, 375 (1–3), May 2007. doi:10.1016/j.tcs.2006.12.026.

Bart Jacobs. *Categorical Logic and Type Theory*. Studies in Logic and The Foundations of Mathematics. Elsevier, 1999. ISBN 0-444-50853-8.

Limin Jia and David Walker. Modal proofs as distributed programs. In *European Symposium on Programming (ESOP)*, 2004. doi:10.1007/978-3-540-24725-8_16.

Limin Jia and Steve Zdancewic. Encoding information flow in Aura. In *Programming Languages and Analysis for Security (PLAS)*, 2009. doi:10.1145/1554339.1554344.

Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Schorr, Joseph, and Steve Zdancewic. Aura: A programming language for authorization and audit. In *International Conference on Functional Programming (ICFP)*, 2008. doi:10.1145/1411204.1411212.

Trevor Jim. SD3: A trust management system with certified evaluation. In *Symposium on Security and Privacy (SSP) (Oakland)*, 2001. doi:10.1109/SECPRI.2001.924291.

Mark P. Jones and Paul Hudak. Implicit and explict parallel programming in haskell. Technical report, Yale University, 1993. URL `http://web.cecs.pdx.edu/~mpj/pubs/par.html`.

Achim Jung and Regina Tix. The troublesome probabilistic powerdomain. In *Workshop on Computation and Approximation (Comprox)*, 1998. doi:10.1016/S1571-0661(05)80216-6.

Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Principles of Programming Languages (POPL)*, 2014. doi:10.1145/2535838.2535846.

Kurt Konolige. *A Deduction Model of Belief*. Morgan Kaufmann, 1986.

Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3), June 1981. doi:10.1016/0022-0000(81)90036-2.

Saul Kripke. A semantical analysis of modal logic I: Normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9, 1963. doi:10.1002/malq.19630090502. Announced in *Journal of Symbolic Logic*, 24:323, 1959.

Joachim Lambek. Deductive systems and categories ii. standard constructions and closed categories. In *Category Theory, Homology Theory and Their Applications I*, 1969. doi:10.1007/BFb0079385.

Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *Symposium on Operating Systems Principles (SOSP)*, 1991. doi:10.1145/138873.138874.

Tom Leinster. General operads and multicategories, 1998. URL `https://arxiv.org/abs/math/9810053`.

Chris Lesniewski-Laas, Bryan Ford, Jacob Strauss, Robert Morris, and M. Frans Kaashoek. Alpaca: extensible authorization for distributed services. In *Computer and Communication Security (CCS)*, 2007. doi:10.1145/1315245.1315299.

Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In *Typed Lambda Calculi and Applications (TLCA)*, 1999. doi:10.1007/3-540-48959-2_17.

Paul Blain Levy. *Call-By-Push-Value*. PhD thesis, Queen Mary and Westfield College University of London, March 2001. URL `http://www.cs.bham.ac.uk/~pbl/papers/thesisqmwphd.pdf`.

Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. A practically implementable and tractable delegation logic. In *Symposium on Security and Privacy (SSP) (Oakland)*, 2000. doi:10.1109/SECPRI.2000.848444.

Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. In *Symposium on Security and Privacy (SSP) (Oakland)*, 2002. doi:10.1109/SECPRI.2002.1004366.

Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Principles of Programming Languages (POPL)*, 2005. doi:10.1145/1040305.1040319.

Peng Li and Steve Zdancewic. Encoding information flow in haskell. In *Computer Security Foundations Workshop (CSFW)*, 2006. doi:10.1109/CSFW.2006.13.

Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Principles and Practice of Declarative Programming (PPDP)*, 2007. doi:10.1145/1273920.1273947.

J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages (POPL)*, 1988. doi:10.1145/73560.73564.

Christoph Lüth and Neil Ghani. Composing monads using coproducts. In *International Conference on Functional Programming (ICFP)*, 2002. doi:10.1145/581478.581492.

Saunders Mac Lane. Natural associativiy and commutativity. *Rice University Studies*, 49(4), September 1963. URL `http://hdl.handle.net/1911/62865`.

Heiko Mantel and David Sands. Controlled Declassification based on Intransitive Noninterference. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2004. doi:10.1007/978-3-540-30477-7_9.

John Maraist, Martin Odersky, David Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. In *Mathematical Foundations of Programming Semantics (MFPS)*, 1995. doi:10.1016/S1571-0661(04)00022-2.

Daniel Marino and Todd Milstein. A generic type-and-effect system. In *Types in Language Design and Implementation (TLDI)*, 2009. doi:10.1145/1481861.1481868.

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

Conor McBride. I got plenty o' nuttin'. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of his 60th Birthday*. Springer, 2016. doi:10.1007/978-3-319-30936-1_12.

Paul André Melliès and Nicolas Tabareau. Resource modalities in game semantics. In *Logic in Computer Science (LICS)*, 2007. doi:10.1109/LICS.2007.41.

Matthew P. Milano and Andrew C. Myers. MixT: A language for mixing consistency in geodistributed transactions. In *Programming Languages Design and Implementation (PLDI)*, 2018. doi:10.1145/3192366.3192375.

Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile

processes, part i. *Information and Computation*, 100(1), September 1992. doi:10.1016/0890-5401(92)90008-4.

Eugenio Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science (LICS)*, 1989. doi:10.1109/LICS.1989.39155.

Fabrizio Montesi. *Choreographic Programming*. PhD thesis, IT University of Copenhagen, 2013. URL `https://www.fabriziomontesi.com/files/choreographic_programming.pdf`.

R.C. Moore and G. Hendrix. Computational models of beliefs and the semantics of belief-sentences. Technical Report 187, SRI International, 1979.

Tom Murphy. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon University, 2008. URL `http://www.cs.cmu.edu/~tom7/papers/modal-types-for-mobile-code.pdf`.

Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Symposium on Security and Privacy (SSP) (Oakland)*, 1998. doi:10.1109/SECPRI.1998.674834.

Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security (JCS)*, 14 (2), May 2006. doi:10.3233/JCS-2006-14203.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic (TOCL)*, 5(N), February 2007. doi:10.1016/j.entcs.2016.06.010.

Sara Negri and Jan von Plato. Sequent calculus in natural deduction style. *Journal of Symbolic Logic*, 66(4), March 2001. doi:10.2307/2694976.

Flemming Nielson. Annotated type and effect systems. *ACM Computing Surveys (CSUR)*, 28(2), June 1996. doi:10.1145/234528.234745.

Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*. Springer, 1999. doi:10.1007/3-540-48092-7_6.

Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, Carnegie Mellon University, 1996. URL `https://www.cs.cmu.edu/~rwh/theses/okasaki.pdf`.

Michel Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning*, 1992a. doi:10.1007/BFb0013061.

Michel Parigot. Free deduction: An analysis of "computations" in classical logic. In *Logic Programming*, 1992b. doi:10.1007/3-540-55460-2_27.

Rohit Parikh. Knowledge and the problem of logical omniscience. In *International Symposium on Methodologies for Intelligent Systems (ISMIS)*, 1987.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: Unified static analysis of context dependence. In *International Conference on Automata, Languages, and Programming (ICALP)*, 2012. doi:10.1007/978-3-642-39212-2_35.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: A calculus of context-dependent computation. In *International Conference on Functional Programming (ICFP)*, 2014. doi:10.1145/2628136.2628160.

Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *Principles of Programming Languages (POPL)*, 1993. doi:10.1145/158511.158524.

Frank Pfenning. Structural cut elimination. In *Logic in Computer Science (LICS)*, June 1995. doi:10.1109/LICS.1995.523253.

Andrew Pimlott and Oleg Kiselyov. Soutei, a logic-based trust-management system. In *Functional and Logic Programming Symposium (FLOPS)*, 2006. doi:10.1007/11737414_10.

Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science (TCS)*, 1(2), December 1975. doi:10.1016/0304-3975(75)90017-1.

Gordon Plotkin and Colin Stirling. A framework for intuitionistic modal logics. In *Theoretical Aspects of Reasoning about Knowledge (TARK)*, pages 399–406, 1986. URL `http://tark.org/proceedings/tark_mar19_86/proceedings.html`.

Amir Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science (FOCS)*, 1977. doi:10.1109/SFCS.1977.32.

Jeff Polakow and Christian Skalka. Specifying distributed trust management in LolliMon. In *Programming Languages and Analysis for Security (PLAS)*, 2006. doi:10.1145/1134744.1134753.

John Power and Hiroshi Watanabe. Combining a monad and a comonad. *Theoretical Computer Science (TCS)*, 280(1–2), May 2002. doi:10.1016/S0304-3975(01)00024-X.

Vincent Rajani, Deepak Garg, and Tamara Rezk. On access control, capabilities, their equivalence, and confused deputy attacks. In *Computer Security Foundations (CSF)*, 2016. doi:10.1109/CSF.2016.18.

Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight

information-flow security in haskell. In *Haskell Symposium (HASKELL)*, 2008. doi:10.1145/1411286.1411289.

Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *International Symposium on Software Security (ISSS)*, 2004. doi:10.1007/978-3-540-37621-7_9.

Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Computer Security Foundations Workshop (CSFW)*, 2005. doi:10.1109/CSFW.2005.15.

Amr Sabry and Philip Wadler. A reflection on call-by-value. *Transactions on Programming Languages and Systems (TOPLAS)*, 1997. doi:10.1145/944705.944723.

Harold Schellinx. *The Noble Art of Linear Decorating*. PhD thesis, Unerversiteit van Amsterdam, 1994. URL `https://hdl.handle.net/11245/1.104138`.

Fred B. Schneider. Personal communication, 2013. January 31, 2013.

Fred B. Schneider, Kevin Walsh, and Emin Gün Sirer. Nexus Authorization Logic (NAL): Design rationale and applications. *Transactions on Information and System Security (TISSEC)*, 14(1), June 2011. doi:10.1145/1952982.1952990.

Alex K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994. URL `http://hdl.handle.net/1842/407`.

Emin Gün Sirer, Willem De Bruijin, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *Symposium on Operating Systems Principles (SOSP)*, 2011. doi:10.1145/2043556.2043580.

Morten Heine Sørensen and Paweł Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006. ISBN 9780444520777.

Robert Stalnaker. The problem of logical omniscience, I. *Synthese*, 89(3), December 1991. doi:10.1007/BF00413506.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in haskell. In *Haskell Symposium (HASKELL)*, 2011. doi:10.1145/2034675.2034688.

Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing policy updates in security-typed languages. In *Computer Security Foundations Workshop (CSFW)*, 2006. doi:10.1109/CSFW.2006.17.

M. E. Szabo. Polycategories. *Communications in Algebra*, 3(8), November 1975. doi:10.1080/00927877508822067.

Gaisi Takeuti. *Proof Theory*. Dover Books on Mathematics. Dover Books, 1987. ISBN 0-486-49073-4. Second Edition, republished by Dover Books in 2013. Originally published by North-Holland, Amsterdam.

Ross Tate. The sequential semantics of producer effect systems. In *Principles of Programming Languages (POPL)*, 2013. doi:10.1145/2429069.2429074.

Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in Mathematics: Volume I*, volume 121 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1988a.

Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in Mathematics: Volume II*, volume 123 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1988b. ISBN 9780444703583.

Tsa-ching Tsai, Alejandro Russo, and John Hughes. A library for secure multi-threaded information flow in haskell. In *Computer Security Foundations (CSF)*, 2007. doi:10.1109/CSF.2007.6.

Tarmo Uustalu and Varmo Vene. Signals and comonads. In *Brazilian Symposium on Programming Languages (SBLP)*, 2005. doi:10.3217/jucs-011-07-1311.

Tarmo Uustalu and Varmo Vene. Comonadic notions of computation. In *Coalgebraic Methods in Computer Science (CMCS)*, 2008. doi:10.1016/j.entcs.2008.05.029.

Matthijs Vákár, Ohad Kammar, and Sam Staton. A domain theory for statisical probabilistic programming. In *Principles of Programming Languages (POPL)*, 2019. doi:10.1145/3290349.

Dirk van Dalen. *Logic and Structure*. Springer, 2004. doi:10.1007/978-1-4471-4558-5.

Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security (JCS)*, 4(3), April 1996. doi:10.3233/JCS-1996-42-304.

Philip Wadler. Linear types can change the world! *Programming Concepts and Methods*, 1990. URL `https://homepages.inf.ed.ac.uk/wadler/topics/linear-logic.html#linear-types`.

Philip Wadler. Call-by-value is dual to call-by-name. In *International Conference on Functional Programming (ICFP)*, 2003. doi:10.1145/944705.944723.

Philip Wadler. Propositions as sessions. In *International Conference on Functional Programming (ICFP)*, 2012. doi:10.1145/2364527.2364568.

Philip Wadler and Peter Thiemann. The marriage of effects and monads. In *International Conference on Functional Programming (ICFP)*, 1998. doi:10.1145/289423.289429.

Lucas Waye, Pablo Buiras, Dan King, Stephen Chong, and Alejandro Russo. It's my privilege: Controlling downgrading in DC-labels. In *Security and Trust Management (STM)*, 2015. doi:10.1007/978-3-319-24858-5_13.

Duminda Wijesekera. Constructive modal logics I. *Annals of Pure and Applied Logic*, 50(3), December 1990. doi:10.1016/0168-0072(90)90059-B.

Glynn Winskel. Distributed probabilistic and quantum strategies. In *Mathematical Foundations of Programming Semantics (MFPS)*, 2013. doi:10.1016/j.entcs.2013.09.024.

Glynn Winskel. On probabilistic distributed strategies. In *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, 2015. doi:10.1007/978-3-319-25150-9_6.

Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *Transactions on Computer Systems (TOCS)*, 12(1), February 1994. doi:10.1145/174613.174614.

Edward Z. Yang. Logitext, 2012. URL `http://logitext.mit.edu/main`. Accessed February 19, 2019.

Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Computer Security Foundations Workshop (CSFW)*, 2001. doi:10.1109/CSFW.2001.930133.

Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylva-

nia, USA, 2009. URL http://reports-archive.adm.cs.cmu.edu/anon/2009/CMU-CS-09-122.pdf.

Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. In *Computer Security Foundations Workshop (CSFW)*, 2005. doi:10.1109/CSFW.2005.16.

APPENDIX A

METATHEORY FOR PROC

Here we prove the properties of Proc that make it a well-behaved semantic domain.

A.1 Preservation

Preservation is a straightforward proof by case. The only interesting part of the proof is showing that consumed and produced can be defined syntactically, as shown in Table A.1. These definitions can be shown to have the property that $\Gamma \vdash \rho \dashv \Delta$ implies $\mathsf{consumed}(\rho)$ is precisely the set of channels in Γ, and likewise $\mathsf{produced}(\rho)$ is precisely the set of channels in Δ. Note that this in turn means type preservation also guarantees consumed and produced are preserved by reduction.

A.2 Progress

In order to prove progress for Comp, we first have to specify the values of Proc. A value in Proc is intuitively a process in which every subcomponent is waiting, directly or indirectly, for messages from the "open" input and output channels of the process. We formalize this in Table A.2 and Figure A.1. The judgement $\rho \vdash x \prec y$ says that no reduction can occur with channel y as the cutpoint until a reduction occurs with channel x as the cutpoint. The one exception is reductions involving $x \rightleftharpoons y$, which we address using a separate connected analysis.

A value is then a process in which every channel occurring in the process is necessarily waiting on some open channel of the process, and every connected

Table A.1: Syntactic Definition of consumed, produced, opened, and closed

ρ	$\mathsf{consumed}(\rho)$	$\mathsf{produced}(\rho)$
$\varnothing$	$\{\}$	$\{\}$
$\rho_1 \parallel \rho_2$	$\left(\bigcup_{\mathsf{consumed}(\rho_2)}^{\mathsf{consumed}(\rho_1)}\right) \setminus \mathsf{closed}(\rho_1, \rho_2)$	$\left(\bigcup_{\mathsf{produced}(\rho_2)}^{\mathsf{produced}(\rho_1)}\right) \setminus \mathsf{closed}(\rho_1, \rho_2)$
$y.\mathsf{init}(c)$	$\{\}$	$\{y\}$
$x \rightleftharpoons y$	$\{x\}$	$\{y\}$
$y.\mathsf{send}()$	$\{\}$	$\{y\}$
$y.\mathsf{send}(x)$	$\{x\}$	$\{y\}$
$\mathsf{handle}_{?x}\{\rho\}$	$\mathsf{consumed}(\rho)$	$\mathsf{produced}(\rho)$
$x.\mathsf{req}()$	$\{x\}$	$\{\}$
$x.\mathsf{req}(y)$	$\{x\}$	$\{y\}$
$x.\mathsf{req}(!y_1, !y_2)$	$\{x\}$	$\{y_1, y_2\}$
$\mathsf{supply}_{!y}\{\rho\}$	$\mathsf{consumed}(\rho)$	$\mathsf{produced}(\rho)$

$$\mathsf{opened}(\rho) = \mathsf{consumed}(\rho) \cup \mathsf{produced}(\rho)$$
$$\mathsf{closed}(\rho_1, \rho_2) = \mathsf{opened}(\rho_1) \cap \mathsf{opened}(\rho_2)$$

Table A.2: Formalization of mentioned and connected

ρ	$\mathsf{mentioned}(\rho)$	$\mathsf{connected}(\rho)$
$\varnothing$	$\{\}$	$\{\}$
$\rho_1 \parallel \rho_2$	$\mathsf{mentioned}(\rho_1) \cup \mathsf{mentioned}(\rho_2)$	$\mathsf{connected}(\rho_1) \cup \mathsf{connected}(\rho_2)$
$y.\mathsf{init}(c)$	$\{y\}$	$\{\}$
$x \rightleftharpoons y$	$\{x, y\}$	$\{x, y\}$
$y.\mathsf{send}()$	$\{y\}$	$\{\}$
$y.\mathsf{send}(x)$	$\{x, y\}$	$\{\}$
$\mathsf{handle}_{?x}\{\rho\}$	$\mathsf{mentioned}(\rho)$	$\mathsf{connected}(\rho) \setminus \mathsf{opened}(\rho)$
$x.\mathsf{req}()$	$\{x\}$	$\{\}$
$x.\mathsf{req}(y)$	$\{x, y\}$	$\{\}$
$x.\mathsf{req}(!y_1, !y_2)$	$\{x, y_1, y_2\}$	$\{\}$
$\mathsf{supply}_{!y}\{\rho\}$	$\mathsf{mentioned}(\rho)$	$\mathsf{connected}(\rho) \setminus \mathsf{opened}(\rho)$

$$\frac{\rho_1 \vdash x \prec y}{\rho_1 \parallel \rho_2 \vdash x \prec y}$$

$$\frac{}{y.\mathsf{send}(x) \vdash y \prec x} \qquad \frac{\rho \vdash y \prec z \qquad z \notin \mathsf{opened}(\rho)}{\mathsf{handle}_{?x}\{\rho\} \vdash y \prec z}$$

$$\frac{}{x.\mathsf{req}(y) \vdash x \prec y} \qquad \frac{}{x.\mathsf{req}(!y_1, !y_2) \vdash x \prec y_1} \qquad \frac{}{x.\mathsf{req}(!y_1, !y_2) \vdash x \prec y_2}$$

$$\frac{\rho \vdash x \prec z \qquad z \notin \mathsf{opened}(\rho)}{\mathsf{supply}_{!y}\{\rho\} \vdash x \prec z}$$

$$\frac{}{\rho \vdash x \preceq x} \qquad \frac{\rho \vdash x \prec y}{\rho \vdash x \preceq y} \qquad \frac{\rho \vdash x \preceq y \qquad \rho \vdash y \preceq z}{\rho \vdash x \preceq z}$$

$$\text{A Proc value is a process } \rho \text{ satisfying } \begin{cases} \forall y \in \mathsf{mentioned}(\rho).\ \exists x \in \mathsf{opened}(\rho).\ \rho \vdash x \preceq y \\ \text{and} \\ \mathsf{connected}(\rho) \subseteq \mathsf{opened}(\rho) \end{cases}$$

Figure A.1: Formalization of Proc Values

channel occurring in the process is itself some open channel of the process. To see that this truly captures the concept of value one would expect, consider two cases. First, suppose a value has no inputs or outputs. Then there are necessarily no channels mentioned in the value, so the value must be $\varnothing$. Second, suppose a value has only one output x of type $\mathbb{N}$. Notice that no case of $\rho \vdash x \prec y$ can apply to a channel x with type $\mathbb{N}$. Consequently, x is the only channel that can be mentioned, and the only processes doing so with the right type for x are of the form $x.\mathsf{init}(c)$.

Furthermore, every value is irreducible. To prove this, consider the reduction rules. In each case, one can easily prove that the cutpoint channel y is either $\mathsf{connected}$ or minimal with respect to $\prec$, by which we mean there is no variable x such that $\rho \vdash x \prec y$. Consequently, a reducible process necessarily has an inter-

$$
\begin{array}{lrcl}
\text{Channels} & x, y, z, \ldots & & \\
\text{Constants} & c & ::= & 0 \mid 1 \mid \ldots \\
\text{Processes} & \rho & ::= & \varnothing \\
 & & \mid & \rho_1 \parallel \rho_2 \\
 & & \mid & y.\mathsf{init}(c) \\
 & & \mid & x \rightleftharpoons y \\
 & & \mid & y.\mathsf{send}(x) \\
 & & \mid & y.\mathsf{send}() \\
 & & \mid & \mathsf{handle}_{?x}\{\rho\} \\
 & & \mid & x.\mathsf{req}() \\
 & & \mid & x.\mathsf{req}(y) \\
 & & \mid & x.\mathsf{req}(!y_1, !y_2) \\
 & & \mid & \mathsf{supply}_{!_c y}\{\rho\} \quad (c \geq 1) \\
\text{Types} & \tau & ::= & \mathbb{N} \mid !_c\tau \mid ?\tau \quad (c \geq 1) \\
\text{Contexts} & \Gamma, \Delta, \Xi & ::= & x : \tau, \ldots \\
 & & & \text{(no repeats)} \\
 & & & \text{(unordered)}
\end{array}
$$

Figure A.2: The Syntax of Proc_n

mediate channel that is either connected or does not depend on an open channel of the process.

Finally, to prove progress, one can do induction on the proof that ρ is well-typed to prove that $\prec$ is well-founded. The key induction invariant for this proof is that $\Gamma \vdash \rho \dashv \Delta$ implies that, for all channels x and y in Δ, if $\rho \vdash x \preceq y$ holds then x and y are the same channel. In combination with well-foundedness of $\prec$, this means there are no dependency chains between the output channels of a process. Well-foundedness is important because it implies that if ρ has any intermediate variables with no dependency on an open variable, then at least one of those intermediate variables, say y, is minimal with respect to $\prec$. Because y is intermediate, it must occur in some Ξ context in the proof that ρ is well-typed. Consequently, one can easily find a non-$\parallel$ process ρ_p producing y in parallel with a non-$\parallel$ process ρ_c consuming y. Enumerating through all the possible cases where

$$\frac{}{\vdash \mathbb{N} \leq \mathbb{N}} \qquad \frac{c \geq c' \qquad \vdash \tau \leq \tau'}{\vdash !_c\tau \leq !_{c'}\tau'} \qquad \frac{\vdash \tau \leq \tau'}{\vdash ?\tau \leq ?\tau'} \qquad \frac{\vdash \tau \leq \tau' \qquad \ldots}{\vdash x : \tau, \ldots \leq x : \tau', \ldots}$$

$$\frac{}{\vdash \varnothing \dashv} \qquad \frac{\Gamma \vdash \rho_1 \dashv \Delta, \Xi \qquad \Gamma', \Xi \vdash \rho_2 \dashv \Delta'}{\Gamma, \Gamma' \vdash \rho_1 \parallel \rho_2 \dashv \Delta, \Delta'} \qquad \frac{}{\vdash y.\mathsf{init}(c) \dashv y : \mathbb{N}}$$

$$\frac{}{x : \tau \vdash x \rightleftharpoons y \dashv y : \tau}$$

$$\frac{}{x : \tau \vdash y.\mathsf{send}(x) \dashv y : ?\tau} \qquad \frac{}{\vdash y.\mathsf{send}() \dashv y : ?\tau}$$

$$\frac{!\Gamma, x : \tau_1 \vdash \rho \dashv y : ?\tau_2}{!\Gamma, x : ?\tau_1 \vdash \mathsf{handle}_{?x}\{\rho\} \dashv y : ?\tau_2}$$

$$\frac{}{x : !_1\tau \vdash x.\mathsf{req}() \dashv} \qquad \frac{}{x : !_1\tau \vdash x.\mathsf{req}(y) \dashv y : \tau}$$

$$\frac{}{x : !_{2c}\tau \vdash x.\mathsf{req}(!y_1, !y_2) \dashv y_1 : !_c\tau, y_2 : !_c\tau} \qquad \frac{x_1 : !_{c_1}\tau_1, \cdots \vdash \rho \dashv y : \tau}{x_1 : !_{c_1 c}\tau_1, \cdots \vdash \mathsf{supply}_{!_c y}\{\rho\} \dashv y : !_c\tau}$$

$$\frac{\vdash \Gamma \leq \Gamma' \qquad \Gamma' \vdash \rho \dashv \Delta' \qquad \vdash \Delta' \leq \Delta}{\Gamma \vdash \rho \dashv \Delta}$$

Figure A.3: The Type System of Proc_n

ρ_p and ρ_c can have the same type for y and y is minimal with respect to $\prec$, one can easily see that each case has a corresponding reduction rule that applies, guaranteeing progress.

A.3 Termination

To show that Proc is terminating, we develop a decreasing well-founded measure. However, in order to build this measure, we actually use a modified version of Proc. This version keeps track of how many times a $!\tau$ channel might be used.

$$\textsc{Parallel}\quad \frac{\rho_1 \to \rho_1'}{\rho_1 \parallel \rho_2 \to \rho_1' \parallel \rho_2}$$

$$\textsc{Context}\quad \frac{\rho \to \rho'}{\mathsf{handle}_{?x}\{\rho\} \to \mathsf{handle}_{?x}\{\rho'\}} \qquad \frac{\rho_1 \to \rho_2}{\mathsf{supply}_{!_c z}\{\rho_1\} \to \mathsf{supply}_{!_c z}\{\rho_2\}}$$

$$\textsc{Identity}\quad \frac{y \in \mathsf{consumed}(\rho)}{x \rightleftharpoons y \parallel \rho \to \rho[y \mapsto x]} \qquad \frac{y \in \mathsf{produced}(\rho)}{\rho \parallel y \rightleftharpoons z \to \rho[y \mapsto z]}$$

$$\textsc{Fail}\quad \frac{\begin{array}{c}\rho_x = \{x.\mathsf{req}() \mid x \in \mathsf{consumed}(\rho) \wedge x \neq y\} \\ \rho_z = \{z.\mathsf{send}() \mid z \in \mathsf{produced}(\rho)\}\end{array}}{y.\mathsf{send}() \parallel \mathsf{handle}_{?y}\{\rho\} \to \rho_x \parallel \rho_z}$$

$$\textsc{Succeed}\quad y.\mathsf{send}(x) \parallel \mathsf{handle}_{?y}\{\rho\} \to \rho[y \mapsto x]$$

$$\textsc{Drop}\quad \mathsf{supply}_{!_c y}\{\rho\} \parallel y.\mathsf{req}() \to \{x.\mathsf{req}() \mid x \in \mathsf{consumed}(\rho)\}$$

$$\textsc{Take}\quad \mathsf{supply}_{!_c y}\{\rho\} \parallel y.\mathsf{req}(z) \to \rho[y \mapsto z]$$

$$\textsc{Clone}\quad \frac{\begin{array}{c}\text{for each } x \in \mathsf{consumed}(\rho), \text{ the channels } y_1^x \text{ and } y_2^x \text{ are fresh} \\ \rho_x = \{x.\mathsf{req}(!y_1^x, !y_2^x) \mid x \in \mathsf{consumed}(\rho)\} \\ \rho_{z_1} = \mathsf{supply}_{!_{\lfloor c/2 \rfloor} z_1}\{\rho\,[y \mapsto z_1,\, x \mapsto y_1^x \mid x \in \mathsf{consumed}(\rho)]\} \\ \rho_{z_2} = \mathsf{supply}_{!_{\lfloor c/2 \rfloor} z_2}\{\rho\,[y \mapsto z_2,\, x \mapsto y_2^x \mid x \in \mathsf{consumed}(\rho)]\}\end{array}}{\mathsf{supply}_{!_c y}\{\rho\} \parallel y.\mathsf{req}(!z_1, !z_2) \to \rho_x \parallel \rho_{z_1} \parallel \rho_{z_2}}$$

$$\textsc{Distribute}\quad \frac{y \in \mathsf{consumed}(\rho_2)}{\mathsf{supply}_{!_{c_1} y}\{\rho_1\} \parallel \mathsf{supply}_{!_{c_2} z}\{\rho_2\} \to \mathsf{supply}_{!_{c_2} z}\{\mathsf{supply}_{!_{\lfloor c_1/c_2 \rfloor} y}\{\rho_1\} \parallel \rho_2\}}$$

$$\frac{y \in \mathsf{consumed}(\rho_2)}{\mathsf{supply}_{!_c y}\{\rho_1\} \parallel \mathsf{handle}_{?x}\{\rho_2\} \to \mathsf{handle}_{?x}\{\mathsf{supply}_{!_c y}\{\rho_1\} \parallel \rho_2\}}$$

$$\frac{y \in \mathsf{produced}(\rho_1)}{\mathsf{handle}_{?x}\{\rho_1\} \parallel \mathsf{handle}_{?y}\{\rho_2\} \to \mathsf{handle}_{?x}\{\rho_1 \parallel \mathsf{handle}_{?y}\{\rho_2\}\}}$$

Figure A.4: Annotated Proc_n Reduction Rules

This language, which we call Proc_n, has the syntax presented in Figure A.2 and type system presented in Figure A.3. The only change is that !s and supplies are annotated with a number capping how many times they can be used.

Given a Proc process ρ, one can construct a Proc_n process ρ_n that erases to ρ, meaning dropping its annotations results in ρ. The typing rules are presented to show how annotations can be inferred proceeding from right to left through the

typing proof (computing the maximum of given annotations where necessary). One can simply initialize every ! in the output types of ρ with 1 and then apply this inference process to get a suitable Proc_n process ρ_n.

Now suppose the Proc process ρ reduces in a single step to ρ'. Then one can easily show that, because ρ_n is well-typed, ρ_n can also reduce (using the annotated reduction rules in Figure A.4) to a Proc_n process ρ'_n that erases to ρ'. Since, with a bit of arithmetic, the annotated reduction rules can also be shown to be type-preserving, ρ'_n will also be well-typed. Repeating this process, one can show that every sequence of reduction steps in well-typed Proc processes has a corresponding sequence of reduction steps in well-typed Proc_n processes. Thus, if every such sequence in Proc_n is guaranteed to terminate, then Proc is necessarily terminating as well.

We now develop a well-founded measure for Proc_n, which we show the Proc_n reduction rules strictly decrease. We use the lexicographic ordering of two measures: $|-|_1$ and $|-|_2$. The first caps the number of times intermediate channels can be eliminated. The second caps the number of times suppliers and handlers can be distributed. The two measures are defined in Table A.3.

Every reduction rule except the DISTRIBUTE rules can easily be shown to strictly reduce the first measure.

Table A.3: Measures for Proc_n Processes

ρ	$\lvert\rho\rvert_1$	$\lvert\rho\rvert_2$
$\varnothing$	0	1
$\rho_1 \parallel \rho_2$	$\lvert\rho_1\rvert_1 + \lvert\rho_2\rvert_1$	$\lvert\rho_1\rvert_2 \cdot \lvert\rho_2\rvert_2$
$y.\mathsf{init}(c)$	0	1
$x \rightleftharpoons y$	1	1
$y.\mathsf{send}(x)$	0	1
$y.\mathsf{send}()$	0	1
$\mathsf{handle}_{?x}\{\rho\}$	$\lvert\rho\rvert_1 + 1$	$\lvert\rho\rvert_2 + 1$
$x.\mathsf{req}()$	0	1
$x.\mathsf{req}(y)$	0	1
$x.\mathsf{req}(!y_1, !y_2)$	0	1
$\mathsf{supply}_{!_c y}\{\rho\}$	$c \cdot \lvert\rho\rvert_1 + 2c - 1$	$\lvert\rho\rvert_2 + 1$

The DISTRIBUTE rules, however, can only be shown to preserve the first measure. Fortunately, it is easy to show that the DISTRIBUTE rules strictly reduce the second measure. As a consequence, every reduction of Proc_n processes also reduces the lexicographic order of these measures. This lexicographic order is well-founded, so this implies Proc_n is terminating, which we have already shown in turn implies that Proc is terminating.

A.4 Confluence

Because Proc is terminating, to prove confluence we only need to prove weak confluence. That is, we only have to show that, for any two ways a given process ρ can be reduced a single step, it is possible to apply further reductions to arrive at the same process. This turns out to be easy to prove.

First, note that reduction is conceptually context-insensitive. That is, if the two reduction steps are applied to disjoint parts of the process, then they trivially commute with each other. Thus we only have two consider reductions that overlap with each other.

Second, note that most pairs of reduction steps cannot overlap for one of two reasons. The first reason is that many reduction steps are syntactically distinct. For example, FAIL reduces a send and a handle, whereas TAKE reduces a supply and a req, so the two cannot overlap. The second reason is that Proc is linear. So although it might seem that FAIL and SUCCEED could each reduce the same handle in different ways, this would mean there are two producers for the same

intermediate channel, which provably means the process must be ill-typed.

At this point, the remaining cases necessarily include a Context, Identity, or Distribute as one of the two steps. The proofs involving Context rely on the fact that reducing a process does not change the set of channels it produces or consumes. The proofs involving Identity are trivial. The proofs involving Distribute are straightforward case analyses. By this high-level reasoning, along with the tedious proof work that we have left to the reader, Proc is weakly confluent and, since Proc is terminating, this in turn implies Proc is confluent.

APPENDIX B

THE FULL FLAFOL PROOF SYSTEM

$$\textsc{Ax}\ \frac{}{\Gamma, \varphi \mathbin{@} g \vdash \varphi \mathbin{@} g} \qquad \textsc{Weakening}\ \frac{\Gamma \vdash \psi \mathbin{@} g}{\Gamma, \varphi \mathbin{@} g' \vdash \psi \mathbin{@} g}$$

$$\textsc{Contraction}\ \frac{\Gamma, (\varphi \mathbin{@} g), (\varphi \mathbin{@} g) \vdash \psi \mathbin{@} g'}{\Gamma, \varphi \mathbin{@} g \vdash \psi \mathbin{@} g'}$$

$$\textsc{Exchange}\ \frac{\Gamma, (\varphi \mathbin{@} g_1), (\psi \mathbin{@} g_2), \Gamma' \vdash \chi \mathbin{@} g}{\Gamma, (\psi \mathbin{@} g_2), (\varphi \mathbin{@} g_1), \Gamma' \vdash \chi \mathbin{@} g}$$

$$\textsc{FalseL}\ \frac{}{\Gamma, \mathsf{False} \mathbin{@} g \vdash \varphi \mathbin{@} g \cdot g'} \qquad \textsc{TrueR}\ \frac{}{\Gamma \vdash \mathsf{True} \mathbin{@} g}$$

$$\textsc{AndL}\ \frac{\Gamma, (\varphi \mathbin{@} g), (\psi \mathbin{@} g) \vdash \chi \mathbin{@} g'}{\Gamma, (\varphi \wedge \psi \mathbin{@} g) \vdash \chi \mathbin{@} g'} \qquad \textsc{AndR}\ \frac{\Gamma \vdash \varphi \mathbin{@} g \qquad \Gamma \vdash \psi \mathbin{@} g}{\Gamma \vdash \varphi \wedge \psi \mathbin{@} g}$$

$$\textsc{OrL}\ \frac{\Gamma, \varphi \mathbin{@} g \vdash \chi \mathbin{@} g' \qquad \Gamma, \psi \mathbin{@} g \vdash \chi \mathbin{@} g'}{\Gamma, (\varphi \vee \psi \mathbin{@} g) \vdash \chi \mathbin{@} g'} \qquad \textsc{OrR1}\ \frac{\Gamma \vdash \varphi \mathbin{@} g}{\Gamma \vdash \varphi \vee \psi \mathbin{@} g}$$

$$\textsc{OrR2}\ \frac{\Gamma \vdash \psi \mathbin{@} g}{\Gamma \vdash \varphi \vee \psi \mathbin{@} g}$$

$$\textsc{ImpL}\ \frac{\Gamma \vdash \varphi \mathbin{@} \langle\rangle \qquad \Gamma, \psi \mathbin{@} g \vdash \chi \mathbin{@} g'}{\Gamma, (\varphi \to \psi \mathbin{@} g) \vdash \chi \mathbin{@} g'} \qquad \textsc{ImpR}\ \frac{\Gamma, \varphi \mathbin{@} \langle\rangle \vdash \psi \mathbin{@} g}{\Gamma \vdash \varphi \to \psi \mathbin{@} g}$$

$$\textsc{ForallL}\ \frac{\Gamma, \varphi[x \mapsto t] \mathbin{@} g \vdash \psi \mathbin{@} g'}{\Gamma, (\forall x{:}\sigma.\, \varphi \mathbin{@} g) \vdash \psi \mathbin{@} g'} \qquad \textsc{ForallR}\ \frac{\Gamma \vdash \varphi \mathbin{@} g \qquad x \notin \mathsf{FV}(\Gamma, g)}{\Gamma \vdash \forall x{:}\sigma.\, \varphi \mathbin{@} g}$$

$$\textsc{ExistsL}\ \frac{\Gamma, \varphi \mathbin{@} g \vdash \psi \mathbin{@} g' \qquad x \notin \mathsf{FV}(\Gamma, \psi, g, g')}{\Gamma, (\exists x{:}\sigma.\, \varphi \mathbin{@} g) \vdash \psi \mathbin{@} g'} \qquad \textsc{ExistsR}\ \frac{\Gamma \vdash \varphi[x \mapsto t] \mathbin{@} g}{\Gamma \vdash \exists x{:}\sigma.\, \psi \mathbin{@} g}$$

$$\textsc{SaysL}\ \frac{\Gamma, \varphi \mathbin{@} g \cdot p\langle \ell \rangle \vdash \psi \mathbin{@} g'}{\Gamma, p\ \mathsf{says}_\ell\ \varphi \mathbin{@} g \vdash \psi \mathbin{@} g'} \qquad \textsc{SaysR}\ \frac{\Gamma \vdash \varphi \mathbin{@} g \cdot p\langle \ell \rangle}{\Gamma \vdash p\ \mathsf{says}_\ell\ \varphi \mathbin{@} g}$$

$$\textsc{SelfL}\ \frac{\Gamma, (\varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot g') \vdash \psi \mathbin{@} g''}{\Gamma, (\varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot p\langle \ell \rangle \cdot g') \vdash \psi \mathbin{@} g''} \qquad \textsc{SelfR}\ \frac{\Gamma \vdash \varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot g'}{\Gamma \vdash \varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot p\langle \ell \rangle \cdot g'}$$

$$\textsc{VarL}\ \frac{\begin{array}{c}\Gamma, (\varphi \mathbin{@} g \cdot p\langle \ell' \rangle \cdot g') \vdash \psi \mathbin{@} g'' \\ \Gamma, (\varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot g') \vdash \ell \sqsubseteq \ell' \mathbin{@} g \cdot p\langle \ell' \rangle\end{array}}{\Gamma, (\varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot g') \vdash \psi \mathbin{@} g''}$$

$$\textsc{VarR}\ \frac{\Gamma \vdash \varphi \mathbin{@} g \cdot p\langle \ell' \rangle \cdot g' \qquad \Gamma \vdash \ell' \sqsubseteq \ell \mathbin{@} g \cdot p\langle \ell \rangle}{\Gamma \vdash \varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot g'}$$

$$\textsc{FwdL}\ \frac{\begin{array}{c}\Gamma, (\varphi \mathbin{@} g \cdot q\langle \ell \rangle \cdot g') \vdash \chi \mathbin{@} g'' \\ \Gamma, (\varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot g') \vdash \mathsf{CanRead}(q, \ell) \mathbin{@} g \cdot p\langle \ell \rangle \\ \Gamma, (\varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot g') \vdash \mathsf{CanWrite}(p, \ell) \mathbin{@} g \cdot q\langle \ell \rangle\end{array}}{\Gamma, \varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot g' \vdash \chi \mathbin{@} g''}$$

$$\textsc{FwdR}\ \frac{\begin{array}{c}\Gamma \vdash \varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot g' \\ \Gamma \vdash \mathsf{CanRead}(q, \ell) \mathbin{@} g \cdot p\langle \ell \rangle \\ \Gamma \vdash \mathsf{CanWrite}(p, \ell) \mathbin{@} g \cdot q\langle \ell \rangle\end{array}}{\Gamma \vdash \varphi \mathbin{@} g \cdot q\langle \ell \rangle \cdot g'}$$

$$\textsc{FlowsToRefl}\ \frac{}{\Gamma \vdash \ell \sqsubseteq \ell \mathbin{@} g}$$

$$\textsc{FlowsToTrans}\ \frac{\Gamma \vdash \ell_1 \sqsubseteq \ell_2 \mathbin{@} g \qquad \Gamma \vdash \ell_2 \sqsubseteq \ell_3 \mathbin{@} g}{\Gamma \vdash \ell_1 \sqsubseteq \ell_3 \mathbin{@} g}$$

$$\textsc{CRVar}\ \frac{\Gamma \vdash \mathsf{CanRead}(p, \ell_2)\ @\ g \qquad \Gamma \vdash \ell_1 \sqsubseteq \ell_2\ @\ g}{\Gamma \vdash \mathsf{CanRead}(p, \ell_1)\ @\ g}$$

$$\textsc{CWVar}\ \frac{\Gamma \vdash \mathsf{CanWrite}(p, \ell_2)\ @\ g \qquad \Gamma \vdash \ell_2 \sqsubseteq \ell_1\ @\ g}{\Gamma \vdash \mathsf{CanWrite}(p, \ell_1)\ @\ g}$$

APPENDIX C

COMPATIBLE SUPERCONTEXTS

$$\textsc{CSCRefl}\ \frac{}{\Gamma \ll \Gamma \vdash \varphi \mathbin{@} g} \qquad \textsc{CSCUnion}\ \frac{\Delta_1 \ll \Gamma \vdash \varphi \mathbin{@} g \qquad \Delta_2 \ll \Gamma \vdash \varphi \mathbin{@} g}{\Delta_1 \cup \Delta_2 \ll \Gamma \vdash \varphi \mathbin{@} g}$$

$$\textsc{CSCContraction}\ \frac{\Delta \ll \Gamma, (\varphi \mathbin{@} g), (\varphi \mathbin{@} g) \vdash \psi \mathbin{@} g'}{\Delta \ll \Gamma, \varphi \mathbin{@} g \vdash \psi \mathbin{@} g'}$$

$$\textsc{CSCExchange}\ \frac{\Delta \ll \Gamma, (\varphi \mathbin{@} g_1), (\psi \mathbin{@} g_2), \Gamma' \vdash \chi \mathbin{@} g}{\Delta \ll \Gamma, (\psi \mathbin{@} g_2), (\varphi \mathbin{@} g_1), \Gamma' \vdash \chi \mathbin{@} g}$$

$$\textsc{CSCAndL}\ \frac{\Delta \ll \Gamma, (\varphi \mathbin{@} g), (\psi \mathbin{@} g) \vdash \chi \mathbin{@} g'}{\Delta \ll \Gamma, (\varphi \wedge \psi \mathbin{@} g) \vdash \chi \mathbin{@} g'} \qquad \textsc{CSCAndR1}\ \frac{\Delta \ll \Gamma \vdash \varphi \mathbin{@} g}{\Delta \ll \Gamma \vdash \varphi \wedge \psi \mathbin{@} g}$$

$$\textsc{CSCAndR2}\ \frac{\Delta \ll \Gamma \vdash \psi \mathbin{@} g}{\Delta \ll \Gamma \vdash \varphi \wedge \psi \mathbin{@} g} \qquad \textsc{CSCOrL1}\ \frac{\Delta \ll \Gamma, \varphi \mathbin{@} g \vdash \chi \mathbin{@} g'}{\Delta \ll \Gamma, (\varphi \vee \psi \mathbin{@} g) \vdash \chi \mathbin{@} g'}$$

$$\textsc{CSCOrL2}\ \frac{\Delta \ll \Gamma, \psi \mathbin{@} g \vdash \chi \mathbin{@} g'}{\Delta \ll \Gamma, (\varphi \vee \psi \mathbin{@} g) \vdash \chi \mathbin{@} g'} \qquad \textsc{CSCOrR1}\ \frac{\Delta \ll \Gamma \vdash \varphi \mathbin{@} g}{\Delta \ll \Gamma \vdash \varphi \vee \psi \mathbin{@} g}$$

$$\textsc{CSCOrR2}\ \frac{\Delta \ll \Gamma \vdash \psi \mathbin{@} g}{\Delta \ll \Gamma \vdash \varphi \vee \psi \mathbin{@} g} \qquad \textsc{CSCImpL1}\ \frac{\Delta \ll \Gamma, \psi \mathbin{@} g \vdash \chi \mathbin{@} g'}{\Delta \ll \Gamma, (\varphi \to \psi \mathbin{@} g) \vdash \chi \mathbin{@} g'}$$

$$\textsc{CSCImpL2}\ \frac{\Delta \ll \Gamma \vdash \varphi \mathbin{@} \langle\rangle}{\Delta \ll \Gamma, (\varphi \to \psi \mathbin{@} g) \vdash \chi \mathbin{@} g'} \qquad \textsc{CSCImpR}\ \frac{\Delta \ll \Gamma, \varphi \mathbin{@} \langle\rangle \vdash \psi \mathbin{@} g}{\Delta \ll \Gamma \vdash \varphi \to \psi \mathbin{@} g}$$

$$\textsc{CSCForallL}\ \frac{\Delta \ll \Gamma, \varphi[x \mapsto t] \mathbin{@} g \vdash \psi \mathbin{@} g'}{\Delta \ll \Gamma, (\forall x : \sigma.\, \varphi \mathbin{@} g) \vdash \psi \mathbin{@} g'}$$

$$\textsc{CSCForallR}\ \frac{\Delta \ll \Gamma \vdash \varphi \mathbin{@} g \qquad x \notin \mathsf{FV}(\Gamma, g)}{\Delta \ll \Gamma \vdash \forall x : \sigma.\, \varphi \mathbin{@} g}$$

$$\textsc{CSCExistsL}\ \frac{\Delta \ll \Gamma, \varphi \mathbin{@} g \vdash \psi \mathbin{@} g' \qquad x \notin \mathsf{FV}(\Gamma, \psi, g, g')}{\Delta \ll \Gamma, (\exists x : \sigma.\, \varphi \mathbin{@} g) \vdash \psi \mathbin{@} g'}$$

$$\textsc{CSCExistsR}\ \frac{\Delta \ll \Gamma \vdash \varphi[x \mapsto t] \mathbin{@} g}{\Delta \ll \Gamma \vdash \exists x : \sigma.\, \varphi \mathbin{@} g} \qquad \textsc{CSCSaysL}\ \frac{\Delta \ll \Gamma, \varphi \mathbin{@} g \cdot p\langle \ell \rangle \vdash \psi \mathbin{@} g'}{\Delta \ll \Gamma, p\ \mathsf{says}_\ell\ \varphi \mathbin{@} g \vdash \psi \mathbin{@} g'}$$

$$\textsc{CSCSaysR}\ \frac{\Delta \ll \Gamma \vdash \varphi \mathbin{@} g \cdot p\langle \ell \rangle}{\Delta \ll \Gamma \vdash p\ \mathsf{says}_\ell\ \varphi \mathbin{@} g}$$

$$\textsc{CSCSelfL}\ \frac{\Delta \ll \Gamma, (\varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot g') \vdash \psi \mathbin{@} g''}{\Delta \ll \Gamma, (\varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot p\langle \ell \rangle \cdot g') \vdash \psi \mathbin{@} g''} \qquad \textsc{CSCSelfR}\ \frac{\Delta \ll \Gamma \vdash \varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot g'}{\Delta \ll \Gamma \vdash \varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot p\langle \ell \rangle \cdot g'}$$

$$\textsc{CSCVarL}\ \frac{\begin{array}{c}\Delta \ll \Gamma, (\varphi \mathbin{@} g \cdot p\langle \ell' \rangle \cdot g') \vdash \psi \mathbin{@} g'' \\ \Gamma, (\varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot g') \vdash \ell \sqsubseteq \ell' \mathbin{@} g \cdot p\langle \ell' \rangle\end{array}}{\Delta \ll \Gamma, (\varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot g') \vdash \psi \mathbin{@} g''}$$

$$\textsc{CSCVarR}\ \frac{\Delta \ll \Gamma \vdash \varphi \mathbin{@} g \cdot p\langle \ell' \rangle \cdot g' \qquad \Gamma \vdash \ell' \sqsubseteq \ell \mathbin{@} g \cdot p\langle \ell \rangle}{\Delta \ll \Gamma \vdash \varphi \mathbin{@} g \cdot p\langle \ell \rangle \cdot g'}$$

$$\textsc{cscfwdl}\ \frac{\begin{array}{c}\Delta \ll \Gamma, (\varphi \mathbin{@} g \cdot q\langle\ell\rangle \cdot g') \vdash \chi \mathbin{@} g'' \\ \Gamma, (\varphi \mathbin{@} g \cdot p\langle\ell\rangle \cdot g') \vdash \mathsf{CanRead}(q, \ell) \mathbin{@} g \cdot p\langle\ell\rangle \\ \Gamma, (\varphi \mathbin{@} g \cdot p\langle\ell\rangle \cdot g') \vdash \mathsf{CanWrite}(p, \ell) \mathbin{@} g \cdot q\langle\ell\rangle\end{array}}{\Delta \ll \Gamma, \varphi \mathbin{@} g \cdot p\langle\ell\rangle \cdot g' \vdash \chi \mathbin{@} g''}$$

$$\textsc{cscfwdr}\ \frac{\begin{array}{c}\Delta \ll \Gamma \vdash \varphi \mathbin{@} g \cdot p\langle\ell\rangle \cdot g' \\ \Gamma \vdash \mathsf{CanRead}(q, \ell) \mathbin{@} g \cdot p\langle\ell\rangle \qquad \Gamma \vdash \mathsf{CanWrite}(p, \ell) \mathbin{@} g \cdot q\langle\ell\rangle\end{array}}{\Delta \ll \Gamma \vdash \varphi \mathbin{@} g \cdot q\langle\ell\rangle \cdot g'}$$