Strict and Lazy Semantics for Effects

Layering Monads and Comonads

ANDREW K. HIRSCH, Cornell University, USA
ROSS TATE, Cornell University, USA

Two particularly important classes of effects are those that can be given semantics using a monad and those that can be given semantics using a comonad. Currently, programs with both kinds of effects are usually given semantics using a technique that relies on a distributive law. While it is known that not every pair of a monad and a comonad has a distributive law, it was previously unknown if there were any realistic pairs of effects that could not be given semantics in this manner. This paper answers that question by giving an example of a pair of effects that cannot be given semantics using a distributive law. Our example furthermore is intimately tied to the duality of strictness and laziness. We discuss how to view this duality through the lens of effects.

CCS Concepts: • **Theory of computation → Categorical semantics**; *Linear logic*; *Denotational semantics*; Proof theory; Type theory;

Additional Key Words and Phrases: monad, comonad, producer effect, consumer effect, layering, distributive law, strictness, laziness, linear logic, classical logic

ACM Reference Format:
Andrew K. Hirsch and Ross Tate. 2018. Strict and Lazy Semantics for Effects: Layering Monads and Comonads. *Proc. ACM Program. Lang.* 2, ICFP, Article 88 (September 2018), 30 pages. https://doi.org/10.1145/3236783

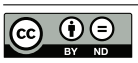
1 INTRODUCTION

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 Millstein 2009; Moggi 1989; Nielson 1996; Nielson and Nielson 1999; Wadler and Thiemann 2003], and those that can be given semantics using a comonad [Brookes and Geva 1992; Brunel et al. 2014; Petricek et al. 2013, 2014; Uustalu and Vene 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 to the study of domain theory [Brookes and van Stone 1993], they did not find any that corresponded to effects. Consequently, when Gaboardi, Katsumata, Orchard, Breuvart, and Uustalu [2016] studied the semantics of languages with both kinds of effects, they decided to focus on the common case of when a distributive law does exist.

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 being an example of this. This poses a challenge, though, because a distributive law is often used in order to compose such effectful programs.

Authors' addresses: Andrew K. Hirsch, Computer Science, Cornell University, Gates Hall, Ithaca, NY, 14853, USA, akhirsch@cs.cornell.edu; Ross Tate, Computer Science, Cornell University, Gates Hall, Ithaca, NY, 14853, USA, ross@cs.cornell.edu.



2475-1421/2018/9-ART88
https://doi.org/10.1145/3236783

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 specifies a semantics that reflects a *strict* interpretation of programs, and the other specifies a semantics that reflects a *lazy* interpretation of 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 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 paper proceeds as follows:

- Section 2 introduces Comp, which is a minimal language with observationally distinct strict and lazy operational semantics.
- Section 3 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 4 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 5 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 6 applies the two layerings to give categorical semantics to Comp. We show that the choice of layering reflects the choice between strict and 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 4.
- Finally, Section 7 reviews related work on effects, monads and comonads, classical and linear logic, strictness and laziness, and focusing and polarization.

In the technical report [Hirsch and Tate 2018], we generalize our semantic techniques to multiple-input and/or multiple-output languages, and we prove the relevant metatheory of our Proc calculus.

Variables	$x, y, z, \ldots$		
Constants	c	::=	$0 \mid 1 \mid \ldots$
Expressions	e	::=	$c \mid x \mid \textbf{error}$
Statements	s	::=	$x := e$
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)

Fig. 1. Comp Syntax

$$\frac{}{\vdash x := c \dashv x : \mathbb{N}}\ \textsc{Const}$$

$$\frac{}{y : t \vdash x := y \dashv x : t}\ \textsc{Var}$$

$$\frac{}{\vdash x := \textbf{error} \dashv x : t}\ \textsc{Error}$$

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

$$\frac{\Gamma \vdash p \dashv y : t'}{\Gamma, x : t \vdash p \dashv y : t'}\ \begin{matrix}(\textsc{Left})\\ \textsc{Weakening}\end{matrix}$$

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

Fig. 2. Comp Typing Rules

2 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 1, Comp is very simple. We designed it to have only the 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 has an expression assigned to it at most once 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 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. 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 proofs are illuminating, not because the types themselves are. For instance, consider proving $\vdash x := 3; y := \textbf{error} \dashv y : \mathbb{N}$. As part of this we will need to prove $x : \mathbb{N} \vdash y := \textbf{error} \dashv y : \mathbb{N}$, which we cannot prove by directly appealing to the Error rule because the Error rule requires an empty context. So instead, we must first apply another rule to get the following proof:

$$\frac{\dfrac{}{\vdash y := \textbf{error} \dashv y : \mathbb{N}}}{x : \mathbb{N} \vdash y := \textbf{error} \dashv y : \mathbb{N}}$$

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.

$$x := c;\ p^+ \ \rightarrow_s\ p^+[x \mapsto c]$$
$$x := y;\ p^+ \ \rightarrow_s\ p^+[x \mapsto y]$$
$$x := \textbf{error};\ p;\ y := e \ \rightarrow_s\ y := \textbf{error}$$

(a) Strict Comp Reduction Rules

$$p^+;\ x := c \ \rightarrow_\ell\ x := c$$
$$p;\ x := e;\ p';\ y := x \ \rightarrow_\ell\ p;\ p'[x \mapsto e];\ y := e$$
$$p^+;\ x := \textbf{error} \ \rightarrow_\ell\ x := \textbf{error}$$

(b) Lazy Comp Reduction Rules

Fig. 3. Comp Reduction Rules

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:

$$x := 3; y := \textbf{error}; z := x$$

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 $z := \textbf{error}$ 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 $x := 3$ 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 $z := 3$ 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 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 Figure 3. In Figure 3a, we see 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 3b, 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 1996; 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 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

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

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 5, we use this connection to category theory to give formal semantics to effectful languages.

3 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 again the example Comp program

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

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, but y 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. On the other hand, 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.

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

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 natural-number 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, known as (Left) *Contraction*, allows a program to use a variable more than once. This provides 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 2003]. 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 belongs to the class of *consumer effects*. Consumer effects are usually given semantics through a categorical construct called a comonad [Brookes and Geva 1992; Petricek et al. 2013, 2014; Uustalu and Vene 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 choosing not to provide an output, or `Some` v, where v is of type t, representing choosing 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.

4 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 1934, 1935]. 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 sequent calculus [Gentzen 1934, 1935].

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, choice (of quantity) would allow processes to reproduce and exhaust a 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.

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

But in order to be a category, these two rules must 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

$$\textsc{Left Weakening}\ \frac{\Gamma \vdash \Delta}{\Gamma, \varphi \vdash \Delta} \qquad \textsc{Left Contraction}\ \frac{\Gamma, \varphi, \varphi \vdash \Delta}{\Gamma, \varphi \vdash \Delta} \qquad \textsc{Right Weakening}\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash \varphi, \Delta} \qquad \textsc{Right Contraction}\ \frac{\Gamma \vdash \varphi, \varphi, \Delta}{\Gamma \vdash \varphi, \Delta}$$

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 exhaust 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 function from A to 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

$$\textsc{Left Weakening}\ \frac{\Gamma \vdash \Delta}{\Gamma, !\varphi \vdash \Delta} \qquad \textsc{Left Contraction}\ \frac{\Gamma, !\varphi, !\varphi \vdash \Delta}{\Gamma, !\varphi \vdash \Delta} \qquad \textsc{Right Weakening}\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash ?\varphi, \Delta} \qquad \textsc{Right Contraction}\ \frac{\Gamma \vdash ?\varphi, ?\varphi, \Delta}{\Gamma \vdash ?\varphi, \Delta}$$

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]

$$\textsc{Right Promotion}\ \frac{!\Gamma \vdash \varphi, ?\Delta}{!\Gamma \vdash !\varphi, ?\Delta} \qquad \textsc{Left Promotion}\ \frac{!\Gamma, \varphi \vdash ?\Delta}{!\Gamma, ?\varphi \vdash ?\Delta}$$

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 produce or consume them.

Contrast this with the final rules involving the exponentials: the dereliction rules.

$$\textsc{Left Dereliction}\ \frac{\Gamma, \varphi \vdash \Delta}{\Gamma, !\varphi \vdash \Delta} \qquad \textsc{Right Dereliction}\ \frac{\Gamma \vdash \varphi, \Delta}{\Gamma \vdash ?\varphi, \Delta}$$

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.

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

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

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

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]. (We discuss further work following on Girard's polarization technique in Section 7.6). 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 of embedding correspond to the layerings. He was able to embed proofs that contained Cut because the layerings do not rely on a distributive law for composition. Consequently, one can amusingly view classical logic as effectful linear logic, and the next section will talk about effectful languages as abstract systems.

5 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 Millstein 2009; Moggi 1989; Petricek et al. 2013, 2014; Tate 2013; Wadler and Thiemann 2003]. 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.

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

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

Fig. 4. Rules of Effectful Languages

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 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).

5.1 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 4 "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.

5.1.1 Producer Effects. A language with a producer effect is one that, in addition to admitting the rules in Figure 4, 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 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

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

Fig. 5. Rules of Producer Effectful Languages

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 $\mathtt{exec}_\tau : M\tau \xrightarrow{\varepsilon} \tau$ that runs the effects captured in $M\tau$. Thus, if ε is the effect "might throw an error," $\mathtt{exec}_\tau$ 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 5. Any language that admits these rules in addition to the rules in Figure 4 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 $\lfloor \mathtt{id}_\tau \rfloor$. Recall that $\mathtt{id}_\tau : \tau \to \tau$ and that pure programs can be turned into effectful programs, so $\mathtt{id}_\tau : \tau \xrightarrow{\varepsilon} \tau$. Thus, $\lfloor \mathtt{id}_\tau \rfloor$ has signature $\tau \to M\tau$, as desired. For *bind*, we have to do something more complicated: $\mathtt{bind}(f) = \lfloor \mathtt{exec}_{\tau_1} ; f ; \mathtt{exec}_{\tau_2} \rfloor$, 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 $\lfloor p ; q \rfloor = \lfloor p \rfloor ; \mathtt{bind}(\lfloor q \rfloor)$ 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 **D** as follows:

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

Theorem 1. *Let $\mathcal{L}$ be a producer-effectful language, and let* **D** *be a category that gives semantics to the pure programs in $\mathcal{L}$. Let M be a monad on* **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 **D**, *and the function on types M forms a monad on that category. The Kleisli category K_M corresponding to this choice of* **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 **D**. Such a situation is more in line with how Moggi [1989] and Wadler and Thiemann [2003] 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 $\mathcal{L}$.

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

Fig. 6. Rules of Consumer Effectful Languages

This distinction is important, particularly for giving semantics to Comp. Comp's type system 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.

5.1.2 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 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 $\mathsf{coexec}_\tau : \tau \xrightarrow{\varepsilon} C\tau$, 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 6. Any language that admits these rules in addition to the rules in Figure 4 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 $\mathsf{cobind}(f)$ as $\lceil \mathsf{coexec}_{\tau_1}; f; \mathsf{coexec}_{\tau_2} \rceil : C\tau_1 \to C\tau_2$ 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, $\mathsf{cobind}(p)$ 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 CoKleisli 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 $\mathtt{id} : a \to a$ 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 $\mathsf{cobind}(f); g$ 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 CoKleisli category K_C corresponding to this choice of $\mathbf{D}$ and C is isomorphic to the category of consumer-effectful programs in $\mathcal{L}$.

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

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

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

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

Fig. 7. Formalization of Doubly-Effectful Languages

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 [2013, 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 Tate [2013].

5.2 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 7. 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 only subsumption rule in Figure 7 applies to only 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.

5.2.1 Distributive Laws for Doubly-Effectful Languages. A common assumption is that producer-effectful programs can be considered doubly-effectful, as can consumer-effectful programs. We formalize this assumption with the two subsumption rules in Figure 8. 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:

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

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

Fig. 8. Subsumption for Distributive Doubly-Effectful Languages

In fact, there are two candidates for building σ_τ: we could use either $\lfloor\lceil \mathsf{exec}_\tau ; \mathsf{coexec}_\tau \rceil\rfloor$ or $\lceil\lfloor \mathsf{exec}_\tau ; \mathsf{coexec}_\tau \rfloor\rceil$. 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 8):

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

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

$$\begin{array}{lllll} \forall q, q' : \tau \xrightarrow{\vec{\varepsilon}} M\tau'. & \varepsilon_p \notin \vec{\varepsilon} & \implies & q ; \mathsf{exec}_{\tau'} = q' ; \mathsf{exec}_{\tau'} & \implies \quad q = q' \\ \forall q, q' : C\tau \xrightarrow{\vec{\varepsilon}} \tau'. & \varepsilon_c \notin \vec{\varepsilon} & \implies & \mathsf{coexec}_\tau ; q = \mathsf{coexec}_\tau ; q' & \implies \quad q = q' \end{array}$$

which are provable from the rules in Figure 7:

$$\begin{array}{lll} q ; \mathsf{exec}_{\tau'} = q' ; \mathsf{exec}_{\tau'} & \implies & q = \lfloor q ; \mathsf{exec}_{\tau'} \rfloor = \lfloor q' ; \mathsf{exec}_{\tau'} \rfloor = q' \\ \mathsf{coexec}_\tau ; q = \mathsf{coexec}_\tau ; q' & \implies & q = \lceil \mathsf{coexec}_\tau ; q \rceil = \lceil \mathsf{coexec}_\tau ; q' \rceil = q' \end{array}$$ □

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

5.2.2 Semantics Based on Distributive Laws. In order to build a distributive law this way, the doubly-effectful language must admit the rules in Figure 8. To see why, consider the composition $\mathsf{exec}_\tau ; \mathsf{coexec}_\tau$. By the rules of Figure 7, two programs can be composed only when their effect sets are the same. In order to compose exec_τ with coexec_τ, we must lift exec_τ from a producer-effectful program to a doubly-effectful program, and similarly with coexec_τ. The subsumption laws of Figure 8 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 8 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 is necessary to build the composition operator for this category. 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 CoKleisli categories, and get $\mathsf{cobind}([f]) : Ca \to CMb$ and $\mathsf{bind}([g]) : MCb \to Mc$, 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 $\mathtt{cobind}([f]);\sigma_b;\mathtt{bind}([g])$. This leads to the following definition:

- 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 8, 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 8.

5.2.3 Linear Logic Lacks a Distributive Law. Girard's attempt to embed classical-logic proofs into linear logic embeds sequents of the form $\Gamma \vdash \Delta$ in classical logic as sequents of the form $!\Gamma \vdash ?\Delta$ in classical linear logic. 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 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.

5.2.4 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

[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 the study of domain theory.

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

Fig. 9. Subsumption for Strict Doubly-Effectful Languages

language. To see why, recall that $\texttt{exec}_\tau$ for throwing errors examines a 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 8, 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 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 8.

Instead, strict languages can only use the weaker subsumption rules in Figure 9. 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 $\left\lceil p; \texttt{coexec}_{\tau_2} \right\rceil$[4].

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 a Kleisli category. This leads to the following definition:

- 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 $\texttt{id}_a : a \to a$ 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 $f; \texttt{bind}(g)$ in $\mathbf{D}$.

[4]This notation is unambiguous here because the proof of Lemma 1 can be adapted to programs in strict doubly-effectful languages with output types of the form $C\tau$.

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

Fig. 10. Subsumption for Lazy Doubly-Effectful Languages

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 9, and let* **D** *be a category that gives semantics to the pure programs in $\mathcal{L}$. Let M be a monad on* **D** *corresponding to the function on types M with appropriate unit and bind, and let C be a comonad on* **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 $\mathcal{L}$.*

Furthermore, the pure programs in $\mathcal{L}$ form a category **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* **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 9. 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 $\lambda(x, s).\ \mathtt{Some}\ (x, s+1)$ might exist, and is of type $\tau \times S \to \mathtt{Maybe}\ (\tau \times S)$ (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.

5.2.5 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 8 to those of Figure 10 (as opposed to those of Figure 9) for lazy doubly-effectful languages.

We can now thunk doubly-effectful programs of the form $p : \tau_1 \xrightarrow{\varepsilon_c, \varepsilon_p} \tau_2$ as $\lceil\!\lceil p \rceil\!\rceil : CM\tau_1 \to M\tau_2$ using the construction $\lceil \mathtt{exec}_{\tau_1}; p \rceil$[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 **D**, a monad M on **D**, and a comonad C on **D**:

- The objects of $K^C_{C,M}$ are the same as the objects of **D**.
- The morphisms from a to b in $K^C_{C,M}$ are the morphisms from CMa to Mb in **D**.
- The identity morphisms $\mathtt{id}_a : a \to a$ in $K^C_{C,M}$ are defined as ϵ_{Ma} in **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 **D**.

[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 $M\tau$.

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 $[\![p;q]\!] = \texttt{cobind}([\![p]\!]);[\![q]\!]$, 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 10, and let* **D** *be a category that gives semantics to the pure programs in $\mathcal{L}$. Let M be a monad on* **D** *corresponding to the function on types M with appropriate unit and bind, and let C be a comonad on* **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 **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* **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.

5.2.6 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 the technical report [Hirsch and Tate 2018].

5.2.7 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 $[\![p]\!]$ 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 (where $M\epsilon_\tau$ is $\lfloor \texttt{exec}_{C\tau}; \lceil \texttt{id}_\tau \rceil \rfloor$ and $C\eta_\tau$ is $\lceil \lfloor \texttt{id}_\tau \rfloor ; \texttt{coexec}_{M\tau} \rceil$).*

$$\llbracket p \rrbracket ; M\epsilon_{\tau_2} \quad = \quad [p] \quad = \quad C\eta_{\tau_1} ; [\![p]\!]$$

Analogously, for any monad M and comonad C on a category **D** *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.

6 GIVING SEMANTICS TO CHOICE IN COMP

In Section 5, 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 that 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 5 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 [Hirsch and Tate 2018].

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.

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 4, 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 ? exponential. The other obvious choice is π-calculus, which has been

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

with the following syntactic identifications

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

Fig. 11. Proc Syntax

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

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

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

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

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

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

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

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

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

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

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

Fig. 12. Proc Typing Rules

shown to be a computational model for full classical linear logic [Abramsky 1994; Beffara 2006; 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.

6.1.1 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 11. 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, $y.\text{send}(x)$ 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 11.

The typing rules for Proc are presented in Figure 12, 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 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

[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 $\bigotimes \Gamma \vdash \parr \Delta$.

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 $y.\text{init}(c)$, $x \rightleftharpoons y$, and $\varnothing$. The process $y.\text{init}(c)$ 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 $y.\text{send}()$, which sends an empty message on y. To model choosing to produce one output, we use the syntax $y.\text{send}(x)$, 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 $\text{handle}_{?x}\{\rho\}$. 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 $x.\text{req}()$ requests nothing from x, which models a Comp program choosing not to use its input. The process $x.\text{req}(y)$ 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, $x.\text{req}(!y_1, !y_2)$ 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 $\text{supply}_{!y}\{\rho\}$. 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.

6.1.2 Semantics. We formalize the behavior of Proc using the reduction rules in Figure 13. 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 [2006]. As a consequence, they enjoy the properties of progress, preservation, confluence, and termination [Hirsch and Tate 2018]. 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 13 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

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

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

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

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

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

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

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

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

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

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

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

Fig. 13. Proc Reduction Rules

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 produced(ρ) and consumed(ρ), which are formalized in the technical report [Hirsch and Tate 2018]. 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 $\text{handle}_{?y}\{\rho\}$ 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 $\|$ operates on multisets of processes.

The Succeed rule defines what happens when a handler $\text{handle}_{?y}\{\rho\}$ 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 formally represent the consumer and producer effects of Comp. Consequently we can apply the layering techniques from Section 5 to give Comp a categorical semantics using Proc.

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 14 and 15.

The translation in Figure 14 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 $\text{handle}_{?x}\{\rho\}$. Handle implements bind for the ? monad in Proc. By using bind, this translation corresponds to the strict semantics of Comp, always propagating 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 $x := 3;\ y := \textbf{error};\ z := x$:

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

This can reduce on either x or y. Since Proc is confluent [Hirsch and Tate 2018], we can reduce on either channel and get the same result. Choose x. The first thing this process does is reduce $x.\text{send}(x')$ and $\text{handle}_{?x}\{\rho\}$, reducing to $\rho[x \mapsto x']$. We then reduce on y, reducing $y.\text{send}()$ and $\text{handle}_{?y}\{\rho\}$ to $x'.\text{req}()$ and $z.\text{send}()$, thereby propagating the error. Then the supplier of x' receives

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

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

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

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

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

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

Fig. 14. Strict Translation

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

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

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

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

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

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

Fig. 15. Lazy Translation

an empty message, which reduces to nothing, leaving just $z.\mathrm{send}()$. 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 15 corresponds to the comonad-prioritizing layering $!?\Gamma \vdash ?\Delta$, derived from using the Kleisli-like category $K^C_{C,M}$. It composes processes using $\mathrm{supply}_{!x}\{\rho\}$. 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

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

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 $x'.\mathrm{init}(3) \parallel z.\mathrm{send}(x')$, 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 [Hirsch and Tate 2018]. 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 $\Gamma \vdash p \dashv x : t \rightsquigarrow_s\ !\Gamma \vdash \rho \dashv x : ?!t$, and a Comp program $x := e$ translates strictly to a Proc process ν, meaning $\Gamma \vdash x := e \dashv x : t \rightsquigarrow_s\ !\Gamma \vdash \nu \dashv x : ?!t$. 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 [Hirsch and Tate 2018], 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

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

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 $x'.\mathrm{init}(c) \parallel x'.\mathrm{send}(x)$, 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 $x.\mathrm{send}()$, which will match with the handler and altogether reduce to $y.\mathrm{send}()$, because y is necessarily produced by ρ_2. This is the translation of $y := \mathbf{error}$.

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 $x := e$, and similarly for the lazy semantics. Because strict Comp is canonicalizing, p must reduce to $x := e'$ for some e', and $x := e'$ 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 $x := e$ and $x := e'$ translate to the same Proc process, which is easily shown to imply that $e = e'$. □

7 RELATED WORK

This paper unifies several threads of work. We review those threads here, focusing on pieces of work not only as individual entities, but also as entities that interact with each other.

7.1 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 [2003] 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; Lüth and Ghani 2002; Peyton Jones and Wadler 1993] and type-and-effect systems [Marino and Millstein 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.

7.2 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 [2013, 2014] identified a way to index comonads. Brunel, Gaboardi, Mazza, and Zdancewic [2014] then adapted Petricek et al.'s work to comonads with weakening and contraction.

7.3 Combining Monads and Comonads

We are not the first to discover the Kleisli-like constructions in Section 5, 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 5. 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.

7.4 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 1996; 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 1996; Zeilberger 2009].

One of these, by Maraist, Odersky, Turner, and Wadler [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 $\mathrm{supply}_{!x}\{-\}$ to force or delay evaluation, whereas we focus on how to compose programs so that they exhibit specific interactions.

7.5 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 1934, 1935].

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 develop and 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.

7.6 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. is that Parigot's logic, Free Deduction [Parigot 1992a], embeds into a unifying logic, LK^{tq}. This allowed them to embed Parigot's $\lambda\mu$-calculus [Parigot 1992b] 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. 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 [1992b], 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.

8 CONCLUSION

In this paper, we gave an example pair of effects to which the currently-preferred semantic technique cannot be applied. This technique requires a distributive law, but no distributive law exists between the monad and comonad representing the example effects. If such a distributive law did exist, because our effects are intimately connected with strictness and laziness, we show that this would imply that strict and lazy semantics do not differ.

In exploring this, we touched upon logic. Our results in this paper allow us to read classical logic as a logic with effects, and classical linear logic as giving semantics to those effects. Indeed, the layering semantics for effects corresponds to preestablished embeddings of classical logic into classical linear logic.

Thus, by combining the perspectives of effects, logic, and semantics, we are able to get new insights into each of these areas. In particular, we found a compositional or categorical interpretation of strictness and laziness that still corresponds to traditional rewrite systems but also bridges to effects and to cut elimination.

ACKNOWLEDGEMENTS

We thank Stephen Brookes, Marco Gaboardi, Guillaume Munch-Maccagnoni, Cornell's Programming Languages Discussion Group, and the anonymous reviewers for their many valuable contributions to this paper.

REFERENCES

Samson Abramsky. 1994. Proofs as Processes. *Theoretical Computer Science* 135, 1 (1994), 5–9.

Jean-Marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation* 2 (1992), 297–347.

Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. 1995. A Call-By-Need Lambda Calculus. In *POPL*. ACM, New York, NY, USA, 233–246.

Robert Atkey. 2009. Parameterised Notions of Computation. *Journal of Functional Programming* 19, 3-4 (2009), 335–376.

Emmanuel Beffara. 2006. A Concurrent Model for Linear Logic. *ENTCS* 155 (2006), 147–168.

G. Bellin and P. J. Scott. 1994. On the π-Calculus and Linear Logic. *Theoretical Computer Science* 135, 1 (1994), 11–65.

Jean Bénabou. 1963. Catégories avec Multiplication. *Comptes Rendus de l'Académie des Sciences Paris* 258 (1963), 771–774.

Stephen Brookes and Shai Geva. 1992. Computational Comonads and Intensional Semantics. In *Applications of Categories in Computer Science*. Cambridge University Press, Cambridge, UK, 1–44.

Stephen Brookes and Kathryn van Stone. 1993. *Monads and Comonads in Intensional Semantics*. Technical Report CMU-CS-93-140. Carnegie Mellon University Department of Computer Science.

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Programming Languages and Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 351–370.

Alonzo Church and J. B. Rosser. 1936. Some Properties of Conversion. *Trans. Amer. Math. Soc.* 39, 3 (1936), 472–482.

Pierre-Louis Curien, Marcelo Fiore, and Guillaume Munch-Maccagnoni. 2016. A Theory of Effects and Resources: Adjunction Models and Polarised Calculi. In *POPL*. ACM, New York, NY, USA, 44–56.

Pierre-Louis Curien and Hugo Herbelin. 2000. The Duality of Computation. In *ICFP*. ACM, New York, NY, USA, 233–243.

Vincent Danos, Jean-Baptiste Joinet, and Harold Schellinx. 1997. A New Deconstructive Logic: Linear Logic. *The Journal of Symbol Logic* 62, 3 (1997), 755–807.

Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication. In *CSL*, Vol. 16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 228–242.

Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. 2014. The Enriched Effect Calculus: Syntax and Semantics. *Journal of Logic and Computation* 24, 3 (2014), 615–654.

Andrzej Filinski. 1999. Representing Layered Monads. In *POPL*. ACM, New York, NY, USA, 175–188.

Andrzej Filinski. 2010. Monads in Action. In *POPL*. ACM, New York, NY, USA, 483–494.

Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining Effects and Coeffects via Grading. In *ICFP*. ACM, New York, NY, USA, 476–489.

Gerhard Gentzen. 1934. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift* 39 (1934), 176–210.

Gerhard Gentzen. 1935. Untersuchungen über das logische Schließen II. *Mathematische Zeitschrift* 39 (1935), 405–431.

Jean-Yves Girard. 1987. Linear Logic. *Theoretical Computer Science* 50, 1 (1987), 1–101.

Jean-Yves Girard. 1991. *A New Constructive Logic: Classical Logic*. Research Report RR-1443. INRIA.

Michael Hicks, Gavin Bierman, Nataliya Guts, Daan Leijen, and Nikhil Swamy. 2014. Polymonadic Programming. *EPTCS* 153 (2014), 79–99.

Andrew K. Hirsch and Ross Tate. 2018. Strict and Lazy Semantics for Effects: Layering Monads and Comonads. In *ICFP*. ACM, New York, NY, USA, Technical Report.

Martin Hyland, Gordon Plotkin, and John Power. 2006. Combining Effects: Sum and Tensor. *Theoretical Computer Science* 357, 1 (2006), 70–99.

Joachim Lambek. 1969. Deductive Systems and Categories II. Standard Constructions and Closed Categories. In *Category Theory, Homology Theory and Their Applications I*. Springer Berlin Heidelberg, Berlin, Heidelberg, 76–122.

Tom Leinster. 1998. General Operads and Multicategories. (1998).

Paul Blain Levy. 1999. Call-By-Push-Value: A Subsuming Paradigm. In *Typed Lambda Calculus and Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg, 228–243.

Paul Blain Levy. 2001. *Call-By-Push-Value*. Ph.D. Dissertation. Queen Mary and Westfield College University of London, London, UK.

Francisco J. López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. 2007. A Simple Rewrite Notion for Call-time Choice Semantics. In *PPDP*. ACM, New York, NY, USA, 197–208.

John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *POPL*. ACM, New York, NY, USA, 47–57.

Christoph Lüth and Neil Ghani. 2002. Composing Monads using Coproducts. In *ICFP*. ACM, New York, NY, USA, 133–144.

Saunders Mac Lane. 1963. Natural Associativiy and Commutativity. *Rice University Studies* 49, 4 (1963), 28–46.

John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. 1995. Call-by-Name, Call-by-Value, Call-by-Need, and the Linear Lambda Calculus. *ENTCS* 1 (1995), 370–392.

Daniel Marino and Todd Millstein. 2009. A Generic Type-and-Effect System. In *TLDI*. ACM, New York, NY, USA, 39–50.

Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I. *Information and Computation* 100, 1 (1992), 1–40.

Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *LICS*. IEEE, Piscataway Township, NJ, USA, 14–23.

Flemming Nielson. 1996. Annotated Type and Effect Systems. *Comput. Surveys* 28, 2 (1996), 344–345.

Flemming Nielson and Hanne Riis Nielson. 1999. Type and Effect Systems. In *Correct System Design: Recent Insights and Advances*. Springer Berlin Heidelberg, Berlin, Heidelberg, 114–136.

Michel Parigot. 1992a. Free Deduction: An Analysis of "Computations" in Classical Logic. In *Logic Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 361–380.

Michel Parigot. 1992b. $\lambda\mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In *Logic Programming and Automated Reasoning*. Springer Berlin Heidelberg, Berlin, Heidelberg, 190–201.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2013. Coeffects: Unified Static Analysis of Context-Dependence. In *ICALP*. Springer-Verlag, Berlin, Heidelberg, 385–397.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-Dependent Computation. In *ICFP*. ACM, New York, NY, USA, 123–135.

Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *POPL*. ACM, New York, NY, USA, 71–84.

G. D. Plotkin. 1975. Call-By-Name, Call-By-Value, and the λ-Calculus. *Theoretical Computer Science* 1, 2 (1975), 125–159.

John Power and Hiroshi Watanabe. 2002. Combining a Monad and a Comonad. *Theoretical Computer Science* 280, 1 (2002), 137–162.

Amr Sabry and Philip Wadler. 1996. A Reflection on Call-by-Value. In *ICFP*. ACM, New York, NY, USA, 13–24.

Harold Schellinx. 1994. *The Noble Art of Linear Decorating*. Ph.D. Dissertation. Unerversiteit van Amsterdam, Amsterdam, Netherlands.

M. E. Szabo. 1975. Polycategories. *Communications in Algebra* 3, 8 (1975), 663–689.

Ross Tate. 2013. The Sequential Semantics of Producer Effect Systems. In *POPL*. ACM, New York, NY, USA, 15–26.

Tarmu Uustalu and Varmo Vene. 2005. Signals and Comonads. *Journal of Universal Computer Science* 11, 7 (2005), 1310–1326.

Tarmo Uustalu and Varmo Vene. 2008. Comonadic Notions of Computation. *ENTCS* 203, 5 (2008), 263–284.

Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming Concepts and Methods*. North-Holland, Amsterdam, Netherlands, 1–21.

Philip Wadler. 2003. Call-by-Value is Dual to Call-by-Name. In *ICFP*. ACM, New York, NY, USA, 189–201.

Philip Wadler. 2012. Propositions as Sessions. In *ICFP*. ACM, New York, NY, USA, 273–286.

Philip Wadler and Peter Thiemann. 2003. The Marriage of Effects and Monads. *Transactions on Computational Logic* 4, 1 (2003), 1–32.

Noam Zeilberger. 2009. *The Logical Basis of Evaluation Order and Pattern-Matching*. Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.