The Sequential Semantics of Producer Effect Systems

Ross Tate
Cornell University
ross@cs.cornell.edu

Abstract

Effects are fundamental to programming languages. Even the lambda calculus has effects, and consequently the two famous evaluation strategies produce different semantics. As such, much research has been done to improve our understanding of effects. Since Moggi introduced monads for his computational lambda calculus, further generalizations have been designed to formalize increasingly complex computational effects, such as indexed monads followed by layered monads followed by parameterized monads. This succession prompted us to determine the most general formalization possible. In searching for this formalization we came across many surprises, such as the insufficiencies of arrows, as well as many unexpected insights, such as the importance of considering an effect as a small component of a whole system rather than just an isolated feature. In this paper we present our semantic formalization for producer effect systems, which we call a *productor*, and prove its maximal generality by focusing on only sequential composition of effectful computations, consequently guaranteeing that the existing monadic techniques are specializations of productors.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

General Terms Languages, Theory

Keywords Effects, Monads, Effectors, Productors, Thunks

1. Introduction

Effects have been around since the beginning of programming languages. After all, even programs written in the lambda calculus have effects: some programs diverge and some programs disregard their inputs. How these effects interact determines the semantics of a program, leading to strict or lazy evaluation depending on what choices are made. Yet, despite their prevalence, effects and their behavior remain fairly mysterious, especially when one considers how thoroughly data and types have been formalized. We view effects and types as complementary systems, so by improving our understanding of effects to match that of types we hope to broaden the perspective of programming languages as a whole.

Before we go any further, we should define what we mean by an effect, since this term has come to have different meanings in different communities. For some an effect is a classification of a computation determined by some analysis [17, 19, 22, 23, 30–32, 34]. For others an effect is synonymous with a monad [3, 9, 12, 13, 38]. Indeed, Wadler and Thiemann illustrated the intimate connection between the former type-and-effect perspective and the latter computational-effect perspective [38], which we will refine within this paper. What we mean by an effect is a classification of a computation independent of the values of its inputs and outputs. For example, "writes to the heap" is an effect but "writes to the heap only if the input integer is non-zero" is not. Thus, whereas types classify the inputs and outputs of computations, effects classify the internals of computations, forming complementary classification systems. More specifically, we are avoiding imposing the complexity of dependent classification systems in addition to the complexity of reasoning about effects in a language-independent manner.

This informal definition is much broader than what many might consider an effect. For example, "disregards its input" satisfies our definition: though it mentions the input to the computation, the meaning of this classification does not depend on the actual value of that input. Similarly, "uses variable `x`" satisfies our definition. These effects correspond to the use of context. They are often disregarded since many consider freely accessible persistent contexts to be a basic part of programming languages, even though such contexts are not necessarily present in resource-constrained or stack-based languages. When we realized this assumption, we had to reconsider our own notion of effect. What we found is that effects like "uses its input multiple times" can be formalized as *consumer* effects because they reason about how a computation consumes its inputs. In fact, many fundamental of behaviors of programming languages reflect these consumer effects. Unfortunately, in order to keep this paper focused, we must consider consumer effects to be outside of the scope at hand. Instead, we will focus on the much more familiar dual notion that we call *producer* effects.

A producer effect reasons about how a computation produces its outputs. For example, "may throw an exception instead of producing its output" or "may examine and alter the heap before producing its output" are familiar producer effects. Here is where most existing work on effects lies, and so likewise here is where we focus our attention. The reason that producer effects are so common is that they are intimately tied to the notion of thunked computations, meaning computations which have been delayed so that they can be treated as values. Indeed, we will define an effect as a producer effect if all computations with that effect can be thunked as "pure" computations for a domain-specific notion of purity. From this basic property, we will prove that the sequential semantics of all producer effects can be formalized by our framework. As a result, Moggi's monads for his computational lambda calculus [20], Wadler's indexed monads [38], Filinski's layered monads [5], and Atkey's parameterized monads [2] are subsumed by our framework, and we will illustrate how these various formalizations arise from simple assumptions on the producer effects at hand, as well as how those assumptions restrict the kind of information that the effect system can track. As an example, we present an effect system for ensuring shared memory is accessed only in critical regions, the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL'13, January 23–25, 2013, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1832-7/13/01. . . $10.00

semantics of which is guaranteed to be inexpressible by these existing systems because it violates their assumptions. However, we show how the semantics can easily be formalized with our framework to prove that the effect system prevents race conditions.

Just as we restrict our scope to producer effects, we also restrict it to sequential composition of computations. There are many other forms of composition, such as multiplicative composition (for parallelism and for adjacent subexpressions), additive composition (for branching), and coinductive composition (for recursion and for loops). We choose sequential composition because that is where existing research has focused its efforts. We focus on just the one form of composition so that we may address its challenges in full. We believe many of the insights and techniques in this paper can be adapted to other forms of composition, though there is still plenty of interesting work to be done there.

With our semantic framework, which we call *productors*, we provide a powerful solution to the problem of formalizing the sequential semantics of producer effects. This solution is not tied to common assumptions such as freely accessible persistent contexts or higher-order functions, enabling us to delve into more interesting semantic domains as well as narrow down the fundamental aspects of programming languages. Because it is so general, not only does our framework provide a means for specifying the semantics of programming languages or first steps towards improving the extensibility of programming languages, it also provides a meta-language for discussing effects and their properties. We will demonstrate how such a meta-language illuminates the constructs in existing languages, particularly in prior formalizations of effects.

This paper begins by first providing an example of an effect system greatly simplified for sake of exposition (Section 2). Then we present monads and how they formalize the semantics of an effect, albeit in a slightly different light than prior such explanations (Section 3). Afterwards, we adapt this approach to our example effect system, showing where and why monads need to be generalized (Section 4). That concludes our example-driven portion of the paper. We then move on to formalizing effect systems as effectors and effectoids (Section 5). We follow that by formalizing their semantics as productors and productoids (Section 6). With these formalisms established, we illustrate how a variety of existing semantic frameworks are special cases of productors arising from extra assumptions on the structure of the effect system at hand (Section 7). We then show what basic properties of a language with effects guarantee all its effects are producer effects and so formalizable using productors (Section 8). Finally, we remark on higher-level insights, opportunities to expand existing semantic techniques, and further directions to take our research in order to meet our goal of providing an abstract language for discussing and formalizing programming languages (Section 9).

2. Effects for Locking

Effects are a classification of computations independent of the explicit inputs and outputs of those computations. Sometimes effects are superimposed on top of an existing language in order to identify optimization opportunities or potential bugs [19, 31, 32, 34]. Other times effects are integrated into the type system [1]. For example, Java has checked exceptions, and Haskell uses monads in order to combine laziness and side effects. In this paper we will demonstrate how patterns in applications of effects lead to patterns in the semantics of effects. To demonstrate that these are just patterns and not necessarily inherent to effects, we present a toy language that breaks from all of these patterns.

Our toy language Crit, shown in Figure 1, has explicit synchronization and shared memory, limited for simplicity in that there is only one lock and one shared integer. Clearly this is just a subset of a more realistic language with branches, functions, and actual parallelism, as well as a much richer type system. However, our intent is just to address sequential composition, so we strip down the language to sequential composition. Nonetheless, even the semantics of this very simple language cannot be formalized using existing frameworks, at least not in a way that allows the semantics to prove the desired properties of the effect system. In particular, the effect system for this language is intended to prove that whenever a lock is acquired it is subsequently released (without being acquired again beforehand) and every use of shared memory occurs within such a critical region.

$$\frac{}{x : \mathbb{Z}, y : \mathbb{Z} \vdash z := x + y \dashv z : \mathbb{Z} \mid \varepsilon}$$

$$\frac{}{\varnothing \vdash \textbf{acquire} \dashv \varnothing \mid \mathsf{locking}} \qquad \frac{}{\varnothing \vdash \textbf{release} \dashv \varnothing \mid \mathsf{unlocking}}$$

$$\frac{}{\varnothing \vdash x := \textbf{get}() \dashv x : \mathbb{Z} \mid \mathsf{critical}} \qquad \frac{}{x : \mathbb{Z} \vdash \textbf{set}(x) \dashv \varnothing \mid \mathsf{critical}}$$

$$\textit{Prop}\,\frac{\Gamma \vdash p \dashv \Gamma' \mid \varepsilon}{\bar{\Gamma}, \Gamma \vdash p \dashv \bar{\Gamma}, \Gamma' \mid \varepsilon} \qquad \textit{Sub}\,\frac{\Gamma \vdash p \dashv \Gamma' \mid \varepsilon \quad \varepsilon \leq \varepsilon'}{\Gamma \vdash p \dashv \Gamma' \mid \varepsilon'}$$

$$\textit{Seq}^2_{\fatsemi}\,\frac{\Gamma \vdash p \dashv \Gamma' \mid \varepsilon \quad \Gamma' \vdash p' \dashv \Gamma'' \mid \varepsilon' \quad \varepsilon \fatsemi \varepsilon' \mapsto \varepsilon''}{\Gamma \vdash p; p' \dashv \Gamma'' \mid \varepsilon''}$$

Figure 1. A toy language Crit with critical-region effects

$$\textsc{Eff} = \{\varepsilon, \mathsf{locking}, \mathsf{unlocking}, \mathsf{critical}, \mathsf{entrant}\}$$
$$\varepsilon \leq \mathsf{critical}, \mathsf{entrant}$$

↓⨟→	ε	locking	unlocking	critical	entrant
ε	ε	locking	unlocking	critical	entrant
locking	locking	—	entrant	locking	—
unlocking	unlocking	critical	—	—	unlocking
critical	critical	—	unlocking	critical	—
entrant	entrant	locking	—	—	entrant

Figure 2. The effect system for Crit

Before we get into the effect system, let us explain Crit at a high level. The judgement $\Gamma \vdash p \dashv \Gamma' \mid \varepsilon$ indicates that the program p consumes context Γ as its input, produces context Γ' as its output, and has overall effect ε. Note that Crit is linear: every value created is used exactly once. There is one exception: an integer stored to the shared state via **set** can be read multiple times via multiple **get**s. We can do this because it is possible to implement a duplicating function $\mathbb{Z} \multimap !\mathbb{Z}$ in linear type theory. One could make Crit nonlinear by restricting its types to ones which have such a duplicating function. We chose not to because it is orthogonal to the concerns of sequential composition and would make the denotational semantics of Crit unnecessarily complex.

Next, consider the five primitive statements of Crit and their effect. We give addition the effect ε, which we call the basic effect, and which more generally can be used for all effects not related to locking (such as non-determinism or memory allocation). The statement **acquire** has the effect locking because overall it changes the state of the lock from free to occupied. Dually, **release** has the effect unlocking because overall it changes the state of the lock from occupied to free. Lastly, **get** and **set** have the effect critical because they must occur inside a critical region.

Finally, consider the three general-purpose rules. *Prop* allows unused context to propagate through a program *without altering the effect*. While this is not necessary, and indeed is not possible for all effect systems (e.g. low-level stack-based effect systems), we will show how this relates to the notion of strength used in existing

semantic frameworks. *Sub* conveys the notion of subeffects analogous to subtyping. $\mathit{Seq}^2_{\varepsilon}$ specifies how effects sequence statically. These rules can easily be used for other languages, so we give their specifics separately in Figure 2. Note that Figure 2 has an additional effect entrant used to indicate code that enters and leaves a critical section (possibly multiple times), so though the code restores the state of the lock it first needs the lock to be free in order to execute.

This effect system has a number of unusual attributes. First, the effects do not form a lattice. In particular, ε is not a subeffect of every effect. Second, not all effects can be sequenced, not even necessarily with themselves. This means that effects can influence the typability of a program. Third, the order of the effects matters. locking before unlocking is considered entrant but the reverse is not allowed at all. All these attributes have important consequences on the denotational semantics of Crit and on the insufficiency of existing frameworks. But before we discuss those we should present the basics of denotational techniques for effects.

3. Monads

In 1958, Godement invented standard constructions [7], which became known as (Kleisli) triples, which became known as monads. In 1988, Moggi migrated the concept of monads from the category-theory community to the programming-languages semantics community [20]. In 1990, Wadler carried this concept over to the functional-languages community [35, 36], and in 1993 these concepts were realized as monadic programming and added to Haskell to make I/O more convenient and to embrace imperative functional programming [12], blurring the line between imperative programming and pure programming. This progression has made monads the most well known technique for formalizing the semantics of effects. Here we show how a monad formalizes an effect, but in a way that highlights opportunities for generalization.

Propagating an Effect Consider the expression $(64 \div x) + 1$ (using integer division). Focus on the problem that $+$ expects its first argument to be an integer but the $\div$ in the first argument may fail to produce one because $\div$ is a partial operation. In other words, $\div$ has the partial effect. We might represent this by saying that $\div$ has type $\mathbb{Z} \times \mathbb{Z} \xrightarrow{\mathsf{partial}} \mathbb{Z}$. We want to formalize what it means to have the partial effect. Moggi observed that we can do so by modifying the return type of $\div$ [20]. In particular, we can view $\div$ as a function that returns an integer *or* a failure code. We can define an algebraic data type P_{partial} to represent these two cases:

$$P_{\mathsf{partial}}(\tau) = \text{success}(\tau) \mid \text{failure}$$

Then we can formalize $\div$ as a *pure* operation $\mathbb{Z} \times \mathbb{Z} \to P_{\mathsf{partial}}(\mathbb{Z})$.

When a failure occurs, any subsequent computations need to propagate this failure. In particular, if $64 \div x$ fails, then $(64 \div x) + 1$ should fail as well. Thus, $(64 \div x) + 1$ also has the partial effect. We can do this by making the pure computation $\lambda v.v + 1$ from $\mathbb{Z}$ to $\mathbb{Z}$ instead take a $P_{\mathsf{partial}}(\mathbb{Z})$ and return a $P_{\mathsf{partial}}(\mathbb{Z})$. This is achieved with a map operation:

$$\begin{aligned} &map_{\mathsf{partial}} : \forall \alpha, \beta.\ (\alpha \to \beta) \to (P_{\mathsf{partial}}(\alpha) \to P_{\mathsf{partial}}(\beta)) \\ &map_{\mathsf{partial}}(f) = \lambda px.\ \textbf{case}\ px \begin{cases} \text{success}(x) \mapsto \text{success}(f(x)) \\ \text{failure} \quad\ \ \mapsto \text{failure} \end{cases} \end{aligned}$$

Thus map turns a pure computation into one which takes an effectful argument and propagates the effect. In this case, map_{partial} indicates that if a failure code is present then all computation should be bypassed and the failure propagated. Using map we can formalize the semantics of $(64 \div x) + 1$ as $map_{\mathsf{partial}}(\lambda v.v + 1)(64 \div x)$. We map the computation after the effectful operation so that it can take an effectful argument, then pass the effectful result to this mapped computation which propagates the effect. Thus if $64 \div x$ fails so does the entire expression.

This pair of a type constructor $P : \text{TYPE} \to \text{TYPE}$ and a function on computations $map : (\tau \to \tau') \to (P(\tau) \to P(\tau'))$ is called an *(endo)functor* (on the category of types) provided it satisfies a few additional equalities which we do not repeat here, and this is just one part of a monad. In the setting of effects, the type constructor P indicates how a production of the effect can be described as data so that effectful computations can be represented as pure computations with a modified return value, and the function map defines how to propagate the effect through pure computations.

Sequencing Effectful Computations Now consider a slightly more complex example: $(64 \div x) \div y$. Here we are sequencing two effectful computations. We can use the functor representation of the partial effect in order to formalize this expression:

$$map_{\mathsf{partial}}(\lambda v.v \div y)(64 \div x)$$

The type of this formalization, though, is $P_{\mathsf{partial}}(P_{\mathsf{partial}}(\mathbb{Z}))$, since the computation we mapped, namely $\lambda v.v \div y$, also has the partial effect. Although having a doubly partial value allows us to determine which $\div$ failed, in this case we are only concerned with whether *any* $\div$ failed. That is, we want the effect of $(64 \div x) \div y$ to be partial rather than (conceptually) $\mathsf{partial}^2$. We can accomplish this by using a monadic $join$ operation to turn doubly partial values into (singly) partial values:

$$\begin{aligned} &join_{\mathsf{partial}} : \forall \alpha. P_{\mathsf{partial}}(P_{\mathsf{partial}}(\alpha)) \to P_{\mathsf{partial}}(\alpha) \\ &join_{\mathsf{partial}}(ppx) = \textbf{case}\ ppx \begin{cases} \text{success}(\text{success}(x)) \mapsto \text{success}(x) \\ \text{success}(\text{failure}) \quad\ \mapsto \text{failure} \\ \text{failure} \qquad\qquad\quad\ \mapsto \text{failure} \end{cases} \end{aligned}$$

Note that in this example there is only one effect in the whole system, and this is why we can use a single monad. Some semantic frameworks generalize from one effect to multiple effects by using multiple monads, but our semantic framework provides a more general system in which these approaches are special cases.

Making Pure Computations Effectful With the above structures we can sequence a non-empty list of effectful computations interspersed with pure computations together into a single effectful computation. Monads have one more component that enables an *empty* list of effectful computations (from a type to itself) to be turned into a single effectful computation. This structure also enables pure computations to be turned into effectful computations. It is known as the unit of the monad, in this case turning pure values into partial values:

$$\begin{aligned} &unit_{\mathsf{partial}} : \forall \alpha.\ \alpha \to P_{\mathsf{partial}}(\alpha) \\ &unit_{\mathsf{partial}}(x) = \text{success}(x) \end{aligned}$$

While it is important to be able to treat pure computations as effectful computations in order to handle computations uniformly, a system with many effects need only have one effect with a unit in order to accomplish this. Recognizing this is important since it allows effects to express significantly more concepts, as we will demonstrate later.

Coherency The operations for monads have a variety of equalities which must hold, known as the identity and associativity laws. We do not repeat those laws here, rather we convey their significance. Consider the expression $(((64 \div x) \div y) \div z) + 5$. This is essentially five computations sequenced together: 64, $\lambda v.v \div x$, $\lambda v.v \div y$, $\lambda v.v \div z$, and $\lambda v.v + 5$. Using the structures above we can sequence these computations (maintaining order) in a variety of ways depending on how we combine consecutive pairs and whether we turn pure computations into effectful computations or simply use functorial structure to propagate effects through them, but the result should be the same no matter which way is used. We call this concept semantic coherency, and the monad laws are necessary and sufficient to ensure this for sequential composition.

4. Semantics for Locking

With monadic techniques in mind, let us consider how we might define a denotational semantics for Crit. First, we need to choose a semantic domain on which to define all of our operations. Choosing a good semantic domain can enable the semantics to automatically ensure useful properties. In this case, we want to ensure that **acquire**s match up with **release**s one-to-one, and **get**s and **set**s happen only inside critical regions.

Selecting a Semantic Domain To this end, we opt for linear type theory as our semantic domain, since linear type theory already has a notion of counting built into it. We use an abstract type L to denote a free lock and an abstract type C to denote the capability to access shared memory. An abstract operation *acquire* converts a free lock L to a capability C, and an abstract operation *release* does the reverse. An abstract operation *get* fetches the value of the shared state provided a capability C is present, and an abstract operation *set* replaces the value of the shared state provided a capability C is present. Of course, L and C actually stand for $\mathbb{Z}$, but by hiding that information and ensuring every primitive operation consumes and produces one L or C we are guaranteed that every *acquire* matches up with a *release* and all *get*s and *set*s occur in between. These abstract constants and their types are summarized below.

$$\begin{array}{rl} L : \textsc{Type} & C : \textsc{Type} \\ \mathit{acquire} : L \multimap C & \mathit{release} : C \multimap L \\ \mathit{get} : C \multimap C \otimes \mathbb{Z} & \mathit{set} : C \otimes \mathbb{Z} \multimap C \end{array}$$

Representing Effectful Values Next, we must specify how to represent productions of each effect as data. While we could define $P_{\epsilon}(\tau)$ as simply τ, we want to emphasize that ϵ represents all effects not related to locking. For example, ϵ could be non-determinism, in which case $P_{\epsilon}(\tau)$ would be $\mu t.\ \tau \oplus (\tau \,\&\, t)$. As such, we simply assume $P_{\epsilon}(\tau)$ is some monad with operations map_{ϵ}, $join_{\epsilon}$, and $unit_{\epsilon}$, along with an operation $strength_{\epsilon}$ of type $\forall \alpha, \beta.\ \alpha \otimes P_{\epsilon}(\beta) \multimap P_{\epsilon}(\alpha \otimes \beta)$ which we will explain later.

As for the impure effects, their semantics all have the form:

$$T_{S,S'}(\tau) = S \multimap P_{\epsilon}(S' \otimes \tau)$$

$T_{S,S'}$ transforms the state from S to S'. In particular, the impure effects transform between L and C:

$$\begin{array}{rr} P_{\mathsf{locking}} = T_{L,C} & P_{\mathsf{unlocking}} = T_{C,L} \\ P_{\mathsf{critical}} = T_{C,C} & P_{\mathsf{entrant}} = T_{L,L} \end{array}$$

Thus a computation with the locking effect must have access to a (necessarily unique) free lock L, do some computation with effect ϵ, and complete with a (necessarily unique) capability to access shared memory C still available.

Formalizing Primitive Operations Once we have determined how to represent an effect, we can give semantics to the primitive effectful computations. The strategy formalizes the semantics of a judgement $\Gamma \vdash s \dashv \Gamma' \mid \varepsilon$ as a term in linear type theory with the type $(\bigotimes \Gamma) \multimap P_{\varepsilon}(\bigotimes \Gamma')$. With this in mind, we define the semantics of the primitive effectful computations in Figure 3.

Note that the semantic domain guarantees that only one portion of code can hold the lock at a time. Thus, just the fact that these primitives can be expressed in that semantic domain ensures that they preserve that property. This helps a language designer determine how they can safely extend the language, since now they need only check that any potential new constructs are expressible in this domain.

Propagating Effects Next, we have to define how effects propagate through pure computations. Since the impure effects have the same shape, they also use approximately the same map operation:

$$map_{T_{S,S'}} = \lambda f.\lambda g.\lambda s.map_{\epsilon}(\lambda\langle s', x\rangle.\langle s', f(x)\rangle)(g(s))$$

$$\begin{array}{c} [\![x : \mathbb{Z}, y : \mathbb{Z} \vdash z := x + y \dashv z : \mathbb{Z} \mid \epsilon]\!] = \lambda\langle m, n\rangle.unit_{\epsilon}(m+n) \\ [\![\varnothing \vdash \textbf{acquire} \dashv \varnothing \mid \mathsf{locking}]\!] = \lambda\langle\rangle.\lambda l.unit_{\epsilon}(\langle \mathit{acquire}(l), \langle\rangle\rangle) \\ [\![\varnothing \vdash \textbf{release} \dashv \varnothing \mid \mathsf{unlocking}]\!] = \lambda\langle\rangle.\lambda c.unit_{\epsilon}(\langle \mathit{release}(c), \langle\rangle\rangle) \\ [\![\varnothing \vdash x := \textbf{get}() \dashv x : \mathbb{Z} \mid \mathsf{critical}]\!] = \lambda\langle\rangle.\lambda c.unit_{\epsilon}(\mathit{get}(c)) \\ [\![x : \mathbb{Z} \vdash \textbf{set}(x) \dashv \varnothing \mid \mathsf{critical}]\!] = \lambda n.\lambda c.unit_{\epsilon}(\langle \mathit{set}(c, n), \langle\rangle\rangle) \end{array}$$

Figure 3. Semantics of the primitive effectful computations of Crit

Sequencing Effectful Computations The most challenging semantic component is sequencing effectful components. Here is where we have to stray from monads. After all, not all of these effects are monads. For example, there is no $join$ operation for locking. To see why, recall that a locking computation needs a free lock to be available and then removes that free lock, turning it into a capability. Thus, immediately after a locking computation there is no free lock available, so there cannot be another locking computation that needs a free lock.

Fortunately, our effect system is designed to address precisely this problem. In Figure 2, locking ⨟ locking is intentionally undefined so that such programs are disallowed and we do not have to worry about sequencing two locking computations. This suggests how we might generalize monads for sequencing computation. Suppose we had functions $p : A \multimap P_{\varepsilon}(B)$ and $p' : B \multimap P_{\varepsilon'}(C)$ representing effectful computions with *different* effects. Furthermore, suppose $\varepsilon \fatsemi \varepsilon'$ were defined as ε''. Then we should be able to combine p and p' into a function of type $A \multimap P_{\varepsilon''}(C)$ representing a computation from A to C with effect ε''. So far, we can use map to get the following function:

$$\lambda a.map_{\varepsilon}(p')(p(a)) : A \multimap P_{\varepsilon}(P_{\varepsilon'}(C))$$

This function essentially produces a doubly effectful value like with monads, but this time with two *different* effects. So, like with monads, we need some way to turn this doubly effectful value into a (singly) effectful value with just effect ε''. Thus, whenever $\varepsilon \fatsemi \varepsilon'$ is defined as ε'' we need the following operation to sequence the two effectful computations:

$$join_{\varepsilon,\varepsilon'} : \forall \alpha.\ P_{\varepsilon}(P_{\varepsilon'}(\alpha)) \multimap P_{\varepsilon''}(\alpha)$$

For Crit there are four major categories of such sequencings: (1) basic followed by basic, (2) impure followed by basic, (3) basic followed by impure, and (4) impure followed by impure.

The first is simply assumed to be defined as discussed before:

$$join_{\epsilon,\epsilon} = join_{\epsilon}$$

The second is fairly simple to define:

$$join_{T_{S,S'},\epsilon} = \lambda f.\lambda s.join_{\epsilon}(map_{\epsilon}(strength_{\epsilon})(f(s)))$$

Note that this is simply the result of sequencing the linear function $f : S \multimap P_{\epsilon}(S' \otimes P_{\epsilon}(\alpha))$, viewed as a computation with effect ϵ, with $strength_{\epsilon} : S' \otimes P_{\epsilon}(\alpha) \multimap P_{\epsilon}(S' \otimes \alpha)$, also viewed as a computation with effect ϵ. What $strength_{\epsilon}$ is doing is pulling the context of the state S' produced by f *into* the effectful value $P_{\epsilon}(\alpha)$ so that it can be accessed by subsequent computations. Unlike in Haskell, $strength_{\epsilon}$ does not exist for all monads in linear type theory. One notable exception is the error monad: $1 \oplus \tau$. This is why languages with both errors and locks have to have special constructs for handling the case when an error is thrown by a thread holding a lock. Similarly, this is why many languages have constructs for resource-sensitive data in the face of errors, such as C#'s `using` construct for `IDisposable` objects. Also, we were careful to define non-determinism using $\&$ rather than $\otimes$ (which corresponds to the (non-empty) list monad used by parsers) since the former has linear strength whereas the latter does not.

Moving on, the third is only slightly more complex to define:

$$join_{\mathfrak{E},T_{S,S'}} = \lambda\hat{f}.\lambda s.join_{\mathfrak{E}}(map_{\mathfrak{E}}(\lambda\langle s,f\rangle.f(s)))(strength_{\mathfrak{E}}(s,\hat{f})))$$

Here we have an S-state s and $\hat{f}$ is an $\mathfrak{E}$-effectful value containing a function expecting an S, so once again we have to use $strength_{\mathfrak{E}}$ and $map_{\mathfrak{E}}$ to give the contained function f access to the state s. This results in a doubly $\mathfrak{E}$-effectful value, so we use $join_{\mathfrak{E}}$.

The fourth case is the most interesting regarding this paper:

$$join_{T_{S,S'},T_{S',S''}} = \lambda f.\lambda s.join_{\mathfrak{E}}(map_{\mathfrak{E}}(\lambda\langle s',g\rangle.g(s'))(f(s)))$$

Note that it is only defined when the intermediate state S' agrees for both effectful computations, in which case it results in a $T_{S,S''}$-effectful computation. In fact, when we consider just the impure effects we get an instance of Atkey's parameterized monads [2], a connection we will discuss in Section 7.

With these components we can define the semantics of $Seq^2_{\fatsemi}$:

$$\frac{[\![\Gamma \vdash p \dashv \Gamma' \mid \varepsilon]\!] = f \qquad [\![\Gamma' \vdash p' \dashv \Gamma'' \mid \varepsilon']\!] = f' \qquad [\![\varepsilon \fatsemi \varepsilon' \mapsto \varepsilon'']\!] = join_{\varepsilon,\varepsilon'}}{[\![\Gamma \vdash p; p' \dashv \Gamma'' \mid \varepsilon'']\!] = \lambda g.join_{\varepsilon,\varepsilon'}(map_{\varepsilon}(f')(f(g)))}$$

Coercing Effectful Computations Crit has another feature, namely subeffects, that we could not encounter with monads since there was only one effect. Like subtypes, the intuition for subeffects is that there is a way to coerce computations pertaining to the subeffect into ones pertaining to the supereffect. As such, whenever $\varepsilon \leq \varepsilon'$ holds, we need an operation with the following form:

$$coerce_{\varepsilon,\varepsilon'} : \forall\alpha.\ P_{\varepsilon}(\alpha) \multimap P_{\varepsilon'}(\alpha)$$

With Crit, we can define such coercions from the basic effect to an impure effect whenever the input and output states match up, namely for critical and entrant:

$$coerce_{\mathfrak{E},T_{S,S}} = \lambda\hat{x}.\lambda s.strength_{\mathfrak{E}}(s,\hat{x})$$

Using these coercions we can define the semantics of *Sub*:

$$\frac{[\![\Gamma \vdash p \dashv \Gamma' \mid \varepsilon]\!] = f \qquad [\![\varepsilon \leq \varepsilon']\!] = coerce_{\varepsilon,\varepsilon'}}{[\![\Gamma \vdash p \dashv \Gamma' \mid \varepsilon']\!] = \lambda g.coerce_{\varepsilon,\varepsilon'}(f(g))}$$

Propagating Context There is only one rule left to handle for Crit. *Prop* is in a sense orthogonal to the concerns of sequential composition, yet it is so common in languages that we include it for sake of discussion. Suppose we have some function $f : \bigotimes\Gamma \multimap P_{\varepsilon}(\bigotimes\Gamma')$ denoting the semantics of $\Gamma \dashv p \vdash \Gamma' \mid \varepsilon$. We want to extend f so that it propagates the values of an additional context $\bar{\Gamma}$ through the computation. It is easy to define the following function that does at least part of the job:

$$(\bigotimes\bar{\Gamma}) \otimes f : (\bigotimes\bar{\Gamma}) \otimes (\bigotimes\Gamma) \multimap (\bigotimes\bar{\Gamma}) \otimes P_{\varepsilon}(\bigotimes\Gamma)$$

So the issue is that we need to somehow pull the $\bigotimes\bar{\Gamma}$ inside the P_{ε} so that we have the necessary $P_{\varepsilon}((\bigotimes\bar{\Gamma}) \otimes (\bigotimes\Gamma))$.

This issue should look familiar, since when sequencing the impure effects we needed some way to pull state inside of $P_{\mathfrak{E}}$, and so we use the same technique we used there. In particular, we need every effect to have a *strength* operation:

$$strength_{\varepsilon} : \forall\alpha,\beta.\ \alpha \otimes P_{\varepsilon}(\beta) \multimap P_{\varepsilon}(\alpha \otimes \beta)$$

We already assumed such an operation has been defined for $\mathfrak{E}$, so we need only define it for the impure effects. Again, the *strength* operations for the impure effects all have the same form:

$$strength_{T_{S,S'}} = \lambda\langle x,f\rangle.\lambda s.map_{\mathfrak{E}}(swap)(strength_{\mathfrak{E}}(x,f(s)))$$
$$swap = \lambda\langle x,\langle s',y\rangle\rangle.\langle s',\langle x,y\rangle\rangle$$

Note that this *strength* need only be defined for types α that can occur in a context Γ. So, if one were to extend Crit to be non-linear and ensure all Crit types τ had an operation $\tau \multimap !\tau$, then the effects could use a non-linear strength. This way Crit's effect system could be extended with exceptions, so long as one were to carefully track and define the interaction of exceptions with locking. However, since we already need the basic effect to have linear strength, we make no such restriction on the types occurring in Γ here.

$$Seq_{\fatsemi}\ \frac{\Gamma_0 \vdash p_1 \dashv \Gamma_1 \mid \varepsilon_1 \quad \ldots \quad \Gamma_{n-1} \vdash p_n \dashv \Gamma_n \mid \varepsilon_n \qquad [\varepsilon_1,\ldots,\varepsilon_n] \overset{\fatsemi}{\mapsto} \varepsilon}{\Gamma_0 \vdash p_1;\ldots;p_n \dashv \Gamma_n \mid \varepsilon}$$

Figure 4. Assumed typing rule for effect systems

We mentioned that *Prop* is not necessary for sequential composition; it comes from the notion that the context is freely accessible. In low-level stack-based languages, effectful operations may be required for even just accessing the context. Sometimes the context can be extended, but only if the effect is altered as well. For example, in a low-level stack-based language, it is occasionally the responsibility/privilege of the exception handler to clean-up/work-with the stack that existed at the point the exception was thrown. Such a language might have an effect of the form $\mathsf{exc}(\hat{\Gamma})$ where $\hat{\Gamma}$ is the state of the stack at the point the exception was thrown. Denotationally, the effect may be represented as $P_{\mathsf{exc}(\hat{\Gamma})}(\tau) = \tau \oplus \bigotimes\hat{\Gamma}$. Should the stack be extended prior to throwing the exception, then this must be encoded in the effect. As such, this language might have a rule like the following:

$$\frac{\Gamma \vdash p \dashv \Gamma' \mid \mathsf{exc}(\hat{\Gamma})}{\bar{\Gamma},\Gamma \vdash p \dashv \bar{\Gamma},\Gamma' \mid \mathsf{exc}(\bar{\Gamma},\hat{\Gamma})}$$

Thus this effect has no need for strength. We emphasize this example both to illustrate the origin of strength present in many existing formalizations and to illustrate the wide variety of languages our framework is capable of handling.

5. Effectors

While so far we have been focusing on our example language Crit, the intent of this work is to apply to nearly all effect systems. In this section, we introduce *effectors* to formalize effect systems as they pertain to typing sequential composition of programs. In the next section, we then present our semantic framework for effectors. Following that, we show how existing semantic frameworks fit within our own. Finally, we show what common properties of a language and its effect system guarantee that the semantics can be formalized with our framework.

For the sake of formalizing effect systems, we assume the unknown language at hand admits the rule $Seq_{\fatsemi}$ in Figure 4. The language may (and should) admit many other rules as well; we simply need to know that it admits at least $Seq_{\fatsemi}$. The symbol $\overset{\fatsemi}{\mapsto}$ is a relationship from lists of effects to effects, so that $[\varepsilon_1,\ldots,\varepsilon_n] \overset{\fatsemi}{\mapsto} \varepsilon$ indicates that ε is the overall effect of sequencing computations with effects ε_1 through ε_n in that order. $Seq_{\fatsemi}$ informs us that the effects of computations are independent of their specific inputs and outputs, granting the modularity one would expect from an effect system. Thus we do not concern ourselves with some form of dependent effects. Note that we intend $Seq_{\fatsemi}$ to include when n is 0, corresponding to typing the empty program. The following short-

hands may further elucidate the meaning of $\overset{\fatsemi}{\mapsto}$:

$$\begin{array}{ccc} \not\varepsilon \mapsto \varepsilon & \text{means} & [] \overset{\fatsemi}{\mapsto} \varepsilon \\ \varepsilon \leq \varepsilon' & \text{means} & [\varepsilon] \overset{\fatsemi}{\mapsto} \varepsilon' \\ \varepsilon \fatsemi \varepsilon' \mapsto \varepsilon'' & \text{means} & [\varepsilon, \varepsilon'] \overset{\fatsemi}{\mapsto} \varepsilon'' \end{array}$$

$\mathit{Seq}_{\fatsemi}$ may not seem like much to work with, but it actually provides some very useful structure. In particular, it is important to realize that given a sequence of computations, say $[p_1, p_2, p_3]$, the syntax $p_1; p_2; p_3$ could be parsed as a whole or as $(p_1; p_2); p_3$ or as $p_1; (p_2; p_3)$ or even as $(); p_1; (); p_2; (); p_3; ()$ with the empty program interspersed. As such, one can show that, should $[\varepsilon_1, \varepsilon_2] \overset{\fatsemi}{\mapsto} \varepsilon'$ and $[\varepsilon', \varepsilon_3] \overset{\fatsemi}{\mapsto} \varepsilon$ hold, then we can assert that $[\varepsilon_1, \varepsilon_2, \varepsilon_3] \overset{\fatsemi}{\mapsto} \varepsilon$ also holds *without changing the effects of programs*. In light of this, we formalize effect systems as follows:

Definition (Effector)**.** A set EFF along with a relation $\overset{\fatsemi}{\mapsto}$ between List(EFF) and EFF satisfying the following two properties:

$$\begin{array}{cc} \textbf{Identity} & \textbf{Associativity} \\ \forall \varepsilon, \varepsilon'. \begin{array}{c} \varepsilon = \varepsilon' \\ \Downarrow \\ [\varepsilon] \overset{\fatsemi}{\mapsto} \varepsilon' \end{array} & \forall \vec{\varepsilon}_1 ... \vec{\varepsilon}_n, \varepsilon. \begin{array}{c} \exists \varepsilon_1 ... \varepsilon_n . (\forall i. \vec{\varepsilon}_i \overset{\fatsemi}{\mapsto} \varepsilon_i) \wedge [\varepsilon_1 ... \varepsilon_n] \overset{\fatsemi}{\mapsto} \varepsilon \\ \Downarrow \\ \vec{\varepsilon}_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \vec{\varepsilon}_n \overset{\fatsemi}{\mapsto} \varepsilon \end{array} \end{array}$$

The identity rule says that sequencing an effectful computation with no other computations should have (at least) the same effect. The associativity rule says that, should one partition a list of effectful computations and sequence each partition together and then sequence each of those results together, sequencing the original list all together should have (at least) the same overall effect. This formalization may seem rather foreign with respect to existing work on effects. However, when an effector is *semi-strict*, meaning the associativity implication is actually an if-and-only-if, there is an equivalent monoid-like definition more akin to prior work:

Theorem 1. A semi-strict effector can equivalently be defined as an *effectoid*: a set EFF along with a unary relation $\not\varepsilon \mapsto -$, a binary relation $- \leq -$, and a ternary relation $- \fatsemi - \mapsto -$ satisfying:

Identity

$$\forall \varepsilon, \varepsilon'. \quad \begin{array}{c} \exists \varepsilon_\ell .\ \not\varepsilon \mapsto \varepsilon_\ell \wedge \varepsilon_\ell \fatsemi \varepsilon \mapsto \varepsilon' \\ \Updownarrow \\ \varepsilon \leq \varepsilon' \\ \Updownarrow \\ \exists \varepsilon_r .\ \not\varepsilon \mapsto \varepsilon_r \wedge \varepsilon \fatsemi \varepsilon_r \mapsto \varepsilon' \end{array}$$

Associativity

$$\forall \varepsilon_1, \varepsilon_2, \varepsilon_3, \varepsilon. \quad \begin{array}{c} \exists \bar{\varepsilon}.\ \varepsilon_1 \fatsemi \varepsilon_2 \mapsto \bar{\varepsilon} \wedge \bar{\varepsilon} \fatsemi \varepsilon_3 \mapsto \varepsilon \\ \Updownarrow \\ \exists \hat{\varepsilon}.\ \varepsilon_2 \fatsemi \varepsilon_3 \mapsto \hat{\varepsilon} \wedge \varepsilon_1 \fatsemi \hat{\varepsilon} \mapsto \varepsilon \end{array}$$

Reflexive Congruence

$$\begin{array}{lc} \forall \varepsilon. & \varepsilon \leq \varepsilon \\ \forall \varepsilon, \varepsilon'. & \not\varepsilon \mapsto \varepsilon \wedge \varepsilon \leq \varepsilon' \implies \not\varepsilon \mapsto \varepsilon' \\ \forall \varepsilon_1, \varepsilon_2, \varepsilon, \varepsilon'. & \varepsilon_1 \fatsemi \varepsilon_2 \mapsto \varepsilon \wedge \varepsilon \leq \varepsilon' \implies \varepsilon_1 \fatsemi \varepsilon_2 \mapsto \varepsilon' \end{array}$$

Proof. The proof can be found in the technical report [33]. □

Semi-strict effectors are common because compositional type systems are common. Conceptually, if $\mathit{Seq}_{\fatsemi}$ is actually an if-and-only-if, then the effector for the language is semi-strict. Note that $\not\varepsilon \mapsto \varepsilon$ does not necessarily imply ε is an identity in the usual sense for monoids. It simply means that pure computations (particularly the empty program) can be given the ε effect. We call such effects *centric* effects.

The effect system for Crit is defined as a congruently preordered partial monoid. This works because congruently preordered partial monoids are equivalent to *principalled* effectoids. A principalled effector is one where for every list of effects $\vec{\varepsilon}$ such that $\vec{\varepsilon} \overset{\fatsemi}{\mapsto} \varepsilon$ holds for some effect ε there exists a minimal such ε with respect to $\leq$. Principalled effectors are common for the same reasons principalled type systems are common, such as simplifying type checking and type inference.

Note that not all effectors are semi-strict, especially ones used in analyses. For example, an analysis may say that `x := *p` has the read effect and that `*p := x` has the write effect but then give `x := *p; *p := x` the $\not\varepsilon$ effect even though generally those effects would combine into the read-write effect. The analysis is using information not contained in the descriptions read and write, namely that the same value and pointer is used in both heap uses, to reason that the program's semantics factor through the operation coercing pure computions into read-write computations and so can be treated as pure. In general, such effectors typically arise when an analysis uses more detailed reasoning intraprocedurally but coarser reasoning interprocedurally. Thus, while effectoids are extremely common, the generality of effectors is necessary to capture existing effect systems.

6. Productors

At last, we present our semantic framework for sequential composition of effectful computations, which we call a productor. Our goal is to give a denotational semantics to $\mathit{Seq}_{\fatsemi}$ in such a way that all the many possible parsings of $p_1; \ldots; p_n$ are guaranteed to produce the same semantics. Our fundamental assumption is that the semantics of the judgement $\Gamma \vdash p \dashv \Gamma' \mid \varepsilon$ can be represented as a morphism $[\![\Gamma]\!] \to P_\varepsilon([\![\Gamma']\!])$ for some function-on-objects P_ε.

Definition (Productor for an effector $\langle \text{EFF}, \overset{\fatsemi}{\mapsto} \rangle$)**.** A category $\mathbf{Sem}$ with endofunctors $\{\langle P_\varepsilon, \mathit{map}_\varepsilon \rangle\}_{\varepsilon \in \text{EFF}}$ and natural transformations $\{\mathit{join}_{[\varepsilon_1,\ldots,\varepsilon_n],\varepsilon} : P_{\varepsilon_1} \circ \cdots \circ P_{\varepsilon_n} \to P_\varepsilon\}_{[\varepsilon_1,\ldots,\varepsilon_n] \overset{\fatsemi}{\mapsto} \varepsilon}$ such that $\mathit{join}_{[\varepsilon],\varepsilon}$ is always the identity transformation and the following diagram commutes whenever all terms are defined:

$$\begin{array}{ccc} & P_{\varepsilon_1} \circ \cdots \circ P_{\varepsilon_n} & \\ {\scriptstyle \mathit{join}_{\vec{\varepsilon}_1,\varepsilon_1} * \ldots * \mathit{join}_{\vec{\varepsilon}_n,\varepsilon_n}} \nearrow & & \searrow {\scriptstyle \mathit{join}_{[\varepsilon_1,\ldots,\varepsilon_n],\varepsilon}} \\ P_{\vec{\varepsilon}_1} \circ \cdots \circ P_{\vec{\varepsilon}_n} & \xrightarrow[\mathit{join}_{\vec{\varepsilon}_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \vec{\varepsilon}_n,\varepsilon}]{} & P_\varepsilon \end{array}$$

Notice the similarity between the commutativity requirements for productors and the implicational requirements for effectors. The idea is to ensure that the many ways that a sequence of programs may be typed all correspond to the same semantics. In other words, the specific proof used to type check a program should be irrelevant to its semantics. In the technical report [33], we ensure precisely that for the following denotational semantics of $\mathit{Seq}_{\fatsemi}$:

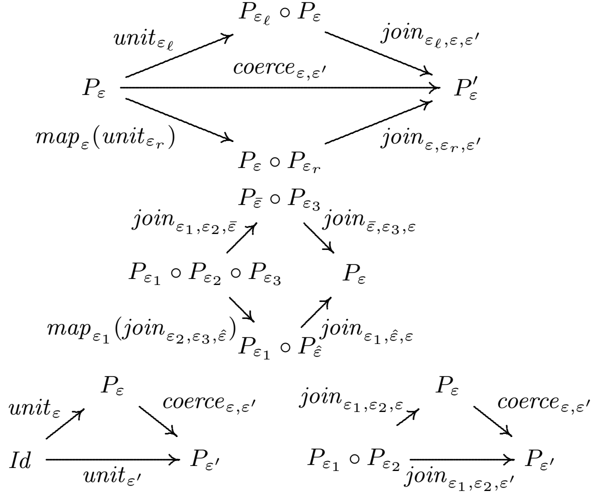
$$\frac{[\![\Gamma_0 \vdash p_1 \dashv \Gamma_1 \mid \varepsilon_1]\!] = f_1 \quad \ldots \quad [\![\Gamma_{n-1} \vdash p_n \dashv \Gamma_n \mid \varepsilon_n]\!] = f_n \qquad \left[\!\!\left[[\varepsilon_1, \ldots, \varepsilon_n] \overset{\fatsemi}{\mapsto} \varepsilon \right]\!\!\right] = \mathit{join}_{[\varepsilon_1,\ldots,\varepsilon_n],\varepsilon}}{[\![\Gamma_0 \vdash p_1; \ldots; p_n \dashv \Gamma_n \mid \varepsilon]\!] = f_1; \mathit{map}_{\varepsilon_1}(\ldots(f_n)); \mathit{join}_{[\varepsilon_1,\ldots,\varepsilon_n],\varepsilon}}$$

One might wonder why we worry about parsing ambiguity. After all, a language could always parse sequential composition left-associatively, like Haskell does. The reason is that by addressing ambiguity concerns we also address the kind of program transformations one would intuitively expect from sequential composition. For example, should $f()$ be defined as $p_1; p_2; p_3$, $g()$ as the empty program, and $h()$ as p_4, then one would expect $f(); g(); h()$ to have the same semantics as $p_1; p_2; p_3; p_4$. This is precisely what our commutative diagrams ensure.

We have proven that when the effector is semi-strict there is an equivalent definition of productors that may appear more familiar for those acquainted with existing semantic frameworks:

Theorem 2. A productor for an effectoid $\langle \text{EFF}, \not\varepsilon, \leq, \fatsemi \rangle$ can equivalently be defined as a *productoid*: a category $\mathbf{Sem}$ with end-

ofunctors $\{\langle P_\varepsilon, map_\varepsilon\rangle\}_{\varepsilon\in\text{EFF}}$ along with natural transformations $\{unit_\varepsilon : \mathbf{Sem} \to P_\varepsilon\}_{\not\varepsilon\mapsto\varepsilon}$ and $\{coerce_{\varepsilon,\varepsilon'} : P_\varepsilon \to P_{\varepsilon'}\}_{\varepsilon\leq\varepsilon'}$ and $\{join_{\varepsilon,\varepsilon',\varepsilon''} : P_\varepsilon \circ P_{\varepsilon'} \to P_{\varepsilon''}\}_{\varepsilon\,⨟\,\varepsilon'\mapsto\varepsilon''}$ such that $coerce_{\varepsilon,\varepsilon}$ is always the identity morphism and the following diagrams commute whenever all terms are defined:



Proof. The proof can be found in the technical report [33]. □

The implicational requirements of an effectoid is essential to the above theorem. Using a productoid for a *pre*-effectoid, meaning an effectoid not necessarily satisfying the relevant implications, can indeed result in ambiguous semantics. Figure 5 provides a toy example of such.

The intuition behind this example is that pure represents no effect, err represents a possible error, and hndl indicates an error handler has been specified (more precisely the $\forall\alpha.\alpha$ must be a natural transformation – note that sometimes there can be many inhabitants of $\forall\alpha.\alpha$, such as differing error messages or even `null` in object-oriented domains). A primitive err-effectful operation **error**() throws an error, and a primitive hndl-effectful operation **handle**(h) sets the handler. The definition of $join_{\mathsf{hndl},\mathsf{hndl},\mathsf{hndl}}$ indicates that, if the handler is set twice in a row, the second handler should be used in place of the first one.

The interaction of handlers and exceptions is defined solely by the $join_{\mathsf{hndl},\mathsf{err},\mathsf{pure}}$ operation. In particular, this definition indicates that, should an exception be thrown, then the previously set handler should be used to handle the exception and proceed with normal execution. The result is that a computation has effect pure only if a handler has been set every place an exception might be thrown.

One can check that these operations do indeed satisfy the requirements of our definition in Theorem 2. However, supposing h_1 and h_2 are handlers, the following pure-effectful program still has two possible semantics:

$$\mathbf{handle}(h_1); \mathbf{handle}(h_2); \mathbf{error}(); \mathbf{error}()$$

In particular, the program can result in h_1 *or* h_2. The program results in h_1 if **handle**(h_2) gets matched up with the first **error**() (via $join_{\mathsf{hndl},\mathsf{err},\mathsf{pure}}$), sequencing into a pure-effectful computation so that then **handle**(h_1) can be matched up with the second **error**() (via $join_{\mathsf{hndl},\mathsf{err},\mathsf{pure}}$), sequencing into a pure-effectful computation resulting in h_1. The program results in h_2 if **handle**(h_2) overrides **handle**(h_1) (via $join_{\mathsf{hndl},\mathsf{hndl},\mathsf{hndl}}$) and the two **error**()s are combined (via $join_{\mathsf{err},\mathsf{err},\mathsf{err}}$), so that then **handle**(h_1); **handle**(h_2) can be matched up with **error**(); **error**(), sequencing into a pure-effectful computation resulting in h_2. Thus, the specific proof used to show the program has effect pure actually influences the program's semantics, violating semantic coherency.

$$\text{EFF} = \{\mathsf{pure}, \mathsf{err}, \mathsf{hndl}\}$$
$$\forall\varepsilon.\quad \mathsf{pure}\,⨟\,\varepsilon \mapsto \varepsilon \quad \varepsilon\,⨟\,\mathsf{pure} \mapsto \varepsilon$$
$$\mathsf{err}\,⨟\,\mathsf{err} \mapsto \mathsf{err} \quad \mathsf{hndl}\,⨟\,\mathsf{hndl} \mapsto \mathsf{hndl} \quad \mathsf{hndl}\,⨟\,\mathsf{err} \mapsto \mathsf{pure}$$

$$P_{\mathsf{pure}}(\tau) = \tau \quad P_{\mathsf{err}}(\tau) = P_{\mathsf{partial}}(\tau) \quad P_{\mathsf{hndl}}(\tau) = \tau \times \forall\alpha.\alpha$$

$$\begin{aligned}
\forall\varepsilon.\quad & join_{\mathsf{pure},\varepsilon,\varepsilon} && = join_{\varepsilon,\mathsf{pure},\varepsilon} = identity \\
& join_{\mathsf{err},\mathsf{err},\mathsf{err}} && = join_{\mathsf{partial}} \\
& join_{\mathsf{hndl},\mathsf{hndl},\mathsf{hndl}} && = \lambda\langle\langle x, h'\rangle, h\rangle.\langle x, h'\rangle \\
& join_{\mathsf{hndl},\mathsf{err},\mathsf{pure}} && = \lambda\langle px, h\rangle.\mathbf{case}\ px \begin{cases} \text{success}(x) \mapsto x \\ \text{failure} \quad\ \mapsto h \end{cases}
\end{aligned}$$

Figure 5. A semantically incoherent pre-effectoid and productoid

This productoid was designed by intentionally devising a pre-effectoid that violates our associativity requirement for effectoids. For example, $\mathsf{hndl}\,⨟\,\mathsf{hndl} \mapsto \mathsf{hndl}$ and $\mathsf{hndl}\,⨟\,\mathsf{err} \mapsto \mathsf{pure}$ hold, but there is no $\hat\varepsilon$ such that $\mathsf{hndl}\,⨟\,\mathsf{err} \mapsto \hat\varepsilon$ and $\mathsf{hndl}\,⨟\,\hat\varepsilon \mapsto \mathsf{pure}$ hold. As such, one cannot progressively transform either proof into the other using the required equivalences on the productoid, so they can produce different semantics.

This illustrates the subtleties of semantic coherency for effect systems addressed by our framework. Because the pre-effectoid does not satisfy the appropriate requirements, the productoid does not extend to a productor. Although not immediately apparent, in general a productor actually has more equational requirements than a productoid in order to maintain semantic coherency; it just happens to be that the implicational requirements of an effectoid imposes enough structure on its productoids to ensure they satisfy those additional equational requirements. In general, additional structure on the effector at hand imposes structure on its productors. This interplay will be a common theme during our discussion of existing semantic frameworks for effects.

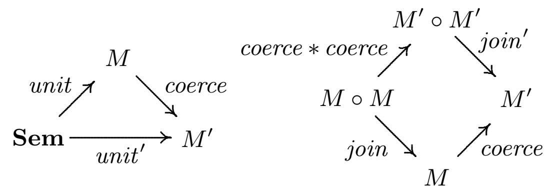
7. Existing Frameworks

While the components and requirements of a productor can be described concisely, the high-level nature of this description can be daunting. As such, it is helpful to see how existing semantic frameworks for effects can be seen as special classes of productors. We start with the simplest such framework: monads.

Monads A monad $\langle M, map, unit, join\rangle$ is a productor for the effector with one effect ε and with $\vec{\varepsilon} \overset{⨟}{\mapsto} \varepsilon$ always holding. M is simply P_ε; map is map_ε; $unit$ is $join_{[],\varepsilon}$; and $join$ is $join_{[\varepsilon,\varepsilon],\varepsilon}$. The equational requirements for a monad are precisely those required by our definition of productor.

More generally, for any effector with an effect ε such that $\not\varepsilon \mapsto \varepsilon$ and $\varepsilon\,⨟\,\varepsilon \mapsto \varepsilon$ hold (called a centric *idempotent* effect), any productor's representation of ε is necessarily a monad.

Monad Morphisms A monad morphism $coerce$ from a monad $\langle M, map, unit, join\rangle$ to another monad $\langle M', map', unit', join'\rangle$ is a natural transformation from M to M' such that the following diagrams commute:



These equational requirements ensure that implicit coercions for effectful computations do not affect the semantics of the program,

in the same spirit as Reynolds' requirements for implicit coercions for data types [28].

As we mentioned, monads correspond to centric idempotent effects. Furthermore, given two such effects ε and ε' with the additional property that ε is a subeffect of ε', then the equational requirements for productors guarantee precisely that the corresponding natural transformation $coerce_{\varepsilon,\varepsilon'} = join_{[\varepsilon],\varepsilon'}$ from P_ε to $P_{\varepsilon'}$ is a monad morphism. Thus both the concepts of monads and monad morphisms are special cases of productors.

Indexed Monads Program analysis is one of the most common applications of effect systems. Such analyses typically use what is known as a type-and-effect system [19, 22, 32], which taints code with a collection of atomic effects. For example, Talpin and Jouvelot taint code with how it accesses various heap regions [31], and Abadi showed that the dependency core calculus essentially takes this approach as well [1]. Abstractly, these effect systems form a join semi-lattice, and the combined effect of a sequence of computations is the join ($\sqcup$) of the effects of the individual computations, with pure and empty computations being given the $\perp$ effect [3, 13].

Wadler and Thiemann famously showed that one of Talpin and Jouvelot's effect systems can be formalized by what has become known as an indexed monad [38]. An indexed monad is a join semi-lattice of monads connected by monad morphisms. The semantics of sequencing effectful computations was defined by coercing both computations to the least common supermonad and then using the $join$ of that monad. They did this for a single effect system, but with our framework it is easy to determine that join semi-lattice effectors are intimately connected to indexed monads, and in these cases the semantics of sequential composition are guaranteed to be defined with this $coerce$-then-$join$ strategy.

To see why, consider the following properties of an effector where sequential composition of effects coincides with a lattice-join operation. They are guaranteed to be *centric* effectors, meaning all their effects are centric effects. They are guaranteed to be *idempotent*, meaning all their effects are idempotent. Lastly, they are guaranteed to be *increasing*, meaning whenever $\vec{\varepsilon} \overset{\fatsemi}{\mapsto} \varepsilon$ holds then all elements of $\vec{\varepsilon}$ are subeffects of ε.

The first two properties already guarantee that a productor represents all effects as monads and furthermore all coercions between such effects must be monad morphisms. Thus a productor for any centric idempotent effector must be a network of monads and monad morphisms, like an indexed monad. All we have left to do is show why sequential composition must take the $coerce$-then-$join$ strategy:

Theorem 3. Any increasing centric idempotent effector is semi-strict. For any productor of an increasing idempotent effector, $join_{\varepsilon_1,\varepsilon_2,\varepsilon}$ is determined by $coerce_{\varepsilon_1,\varepsilon}$, $coerce_{\varepsilon_2,\varepsilon}$, and $join_{\varepsilon,\varepsilon,\varepsilon}$.

Proof. First, suppose $\vec{\varepsilon}_1 +\!\!+ \ldots +\!\!+ \vec{\varepsilon}_n \mapsto \varepsilon$ holds. Since the effector is increasing, all effects in each $\vec{\varepsilon}_i$ must be subeffects of ε. Since the effector is centric and idempotent, $[\varepsilon, \ldots, \varepsilon] \mapsto \varepsilon$ holds for all arities. These and the definition of effector imply that $\vec{\varepsilon}_i \mapsto \varepsilon$ holds. Thus, since $[\varepsilon, \ldots, \varepsilon] \mapsto \varepsilon$ also holds, the effector is semi-strict.

Second, suppose $\varepsilon_1 \fatsemi \varepsilon_2 \mapsto \varepsilon$ holds. Since the effector is increasing, $\varepsilon_i \leq \varepsilon$ holds so $join_{[\varepsilon_i],\varepsilon}$ must exist. Since the effector is idempotent, $[\varepsilon, \varepsilon] \mapsto \varepsilon$ holds so $join_{[\varepsilon,\varepsilon],\varepsilon}$ must exist. Then by definition of productor since $[\varepsilon_1] +\!\!+ [\varepsilon_2]$ equals $[\varepsilon_1, \varepsilon_2]$, $join_{[\varepsilon_1,\varepsilon_2],\varepsilon}$ must equal $(join_{[\varepsilon_1],\varepsilon} * join_{[\varepsilon_2],\varepsilon})\,;join_{[\varepsilon,\varepsilon],\varepsilon}$, which is the long-hand form of $(coerce_{\varepsilon_1,\varepsilon} * coerce_{\varepsilon_2,\varepsilon})\,;join_{\varepsilon,\varepsilon,\varepsilon}$. □

In the other direction, let us consider what kind of effectors indexed monads can handle. Suppose an effector is represented by an indexed monad. Then that effector can be faithfully extended so that it is a principalled, idempotent, increasing, centric, and furthermore *commutative*, meaning if $\vec{\varepsilon} \overset{\fatsemi}{\mapsto} \varepsilon$ holds then $\vec{\epsilon} \overset{\fatsemi}{\mapsto} \varepsilon$ holds for all permutations $\vec{\epsilon}$ of $\vec{\varepsilon}$. This tells us three things: the effector cannot guarantee properties that depend on an effectful computation occurring; the effector cannot guarantee properties that depend on the order of effectful computations; and the effector cannot guarantee properties that depend on the frequency of effectful computations.

First, we know that an effector represented by an indexed monad cannot have effects that guarantee an effectful computation occurs, since all effects are centric. In Crit it is important to distinguish computations that *must* lock from those that *might* lock in order to ensure shared-memory accesses occur in critical regions. In particular, the effector must not confuse a pure or empty computation for one that definitely locks, so that whatever effect ε used to identify definite locking cannot satisfy $\mathfrak{e} \mapsto \varepsilon$ and so cannot be centric. As such, indexed monads cannot be used for effectors that guarantee locking or guarantee responsiveness (e.g. a server always producing a response for each request).

Second, we know that an effector represented by an indexed monad cannot guarantee properties dependent on the order of operations, since the effector is commutative. In Crit it is important to distinguish accessing shared memory *after* acquiring a lock from accessing shared memory *before* acquiring a lock. Yet, if the effector were commutative, then locking ⨾ critical would necessarily equal critical ⨾ locking so such computations would be indistinguishable in the effector. As such, indexed monads cannot be used for effectors that need order sensitivity.

Third, we know that an effector represented by an indexed monad cannot guarantee properties dependent on the frequency of operations, since each effect is idempotent. In Crit it is important to distinguish acquiring a lock once from acquiring a lock multiple times since Crit prevents reentrant code. However, if locking were idempotent, then locking ⨾ locking would have to equal locking so that multiple acquisitions would appear the same as a single acquisition. As such, indexed monads cannot be used for effectors that want to track the quantity of effectful operations.

These limitations of indexed monads were part of why we endeavored to find a more general framework. The discussion here illustrates that, not only are there advantages to having a more general semantic framework, but our formalizations provide a useful language for discussing effectors, another reason why we began our investigations. It also bears some importance for effect analyses, since our findings indicate that lattice-based effectors have some severe limitations in what they can determine about a program. We hope these insights will encourage investigations into new analytical techniques and new semantic techniques.

Layered Monads Filinski introduced the semantic technique of *layered* monads [5]. A layering of a monad $\langle M', map', unit', join'\rangle$ over another monad $\langle M, map, unit, join\rangle$ is a natural transformation $layer : M \circ M' \rightarrow M'$ satisfying certain equational properties. $layer$ looks very much like our $join$ operation for effects $\top$ and $\perp$ should $\perp \fatsemi \top \mapsto \top$ hold. Indeed, Filinski's equational requirements for layerings are simply our equivalent requirements for productoids of an increasing centric idempotent effectoid, and Filinski uses the $layer$ operation to sequence a $\perp$ computation followed by a $\top$ computation, as our framework would do with the corresponding $join$ operation.

In more recent work [6], Filinski allows users to build a tree of monadic subeffects, with non-termination at the root of the tree, so that programmers could build their own effect system. Any effect system designed in such a way is guaranteed to be an increasing centric idempotent effector and so has many of the key limitations that we discussed for indexed monads. However, there is one significant difference: indexed monads necessarily correspond to *total* effectors whereas Filinski's tree of subeffects may not be

total. An effector is total if for every list of effects $\vec{\varepsilon}$ there is some effect ε such that $\vec{\varepsilon} \overset{\fatsemi}{\mapsto} \varepsilon$ holds. This is not necessarily a limitation of Filinski's approach, though. It has the advantage that it can express user effects that are incompatible with each other. This is an important property for a modular language where users should be able to design their effects without worrying about what other effects may be present elsewhere in the system should they never meet each other.

Parameterized Monads Atkey made the observation that many effect systems have to do with transitioning from one state to another [2]. Indeed, most of the effects in Crit are used to indicate the required incoming state of the lock and the guaranteed outgoing state of the lock. As such, Atkey designed a *parameterized* monad to address this common case, and showed how parameterized monads can formalize a variety of concepts such as composable continuations [4, 37]. A parameterized monad is (equivalently)

- a category $\mathbf{S}$ of states and primitive transitioning operations
- a functor $\langle T, \mathit{map}\rangle : \mathbf{S}^{op} \times \mathbf{S} \to (\mathbf{Sem} \to \mathbf{Sem})$
- an extranatural transformation $\mathit{unit} : \forall s.\ \mathit{Id} \to T(s, s)$
- an extranatural transformation
$$\mathit{join} : \forall s, s', s''.\ T(s, s') \circ T(s', s'') \to T(s, s'')$$

satisfying equational requirements we do not repeat here.

This should look slightly familiar. In fact, the productor corresponding to just the impure effects of Crit is a parameterized monad. The category $\mathbf{S}$ is just the discrete category with objects *free* and *occupied*. The extranaturality conditions hold automatically since the category is discrete.

Now, it is important to note that a parameterized monad does two things simultaneously: it specifies how to sequence computations *and* it specifies the semantics of primitive operations. For example, our semantics for **acquire** and **release** could be adapted into a parameterized monad for the free category generated by the graph with two objects and a morphism between them in each direction. However, the intent of our framework is only to formalize sequential composition of computations, so we focus on a restricted subset of parameterizations. In particular, we focus on parameterized monads where $\mathbf{S}$ is thin, meaning there is at most one morphism between any two states. Such an $\mathbf{S}$ is actually a preorder, and so represents states with a substate relation.

Given such a set of states S and a substate preorder $\leq$, define an effectoid $\mathcal{E}_{\langle S, \leq\rangle}$ as follows:

- EFF is $S \times S$, so that the effect $\langle s, s'\rangle$ represents computations that transition from state s to state s'.
- $\varepsilon \mapsto \langle s, s'\rangle$ holds iff $s \leq s'$ holds.
- $\langle s_1', s_2\rangle \leq \langle s_1, s_2'\rangle$ holds iff $s_1 \leq s_1'$ and $s_2 \leq s_2'$ hold.
- $\langle s_1', s_2\rangle \fatsemi \langle s_2', s_3\rangle \mapsto \langle s_1, s_3'\rangle$ holds iff $s_1 \leq s_1'$, $s_2 \leq s_2'$, and $s_3 \leq s_3'$ hold.

Due to their definition, all effectors formalizable by parameterized monads are of this form.

Theorem 4. Given a set of states S and a substate preorder $\leq$, there is a one-to-one correspondence between parameterized monads for the category $\langle S, \leq\rangle$ and productoids for the effectoid $\mathcal{E}_{\langle S, \leq\rangle}$.

Proof. Given a parameterized monad $\langle T, \mathit{map}, \mathit{unit}, \mathit{join}\rangle$ for $\langle S, \leq\rangle$, define the productoid for $\mathcal{E}_{\langle S, \leq\rangle}$ as follows:

- $\langle P_{\langle s, s'\rangle}, \mathit{map}_{\langle s, s'\rangle}\rangle = T(s, s')$
- $\mathit{unit}_{\langle s, s'\rangle} = \mathit{Id} \xrightarrow{\mathit{unit}_s} T(s, s) \xrightarrow{\mathit{map}(s \geq s, s \leq s')} T(s, s')$
- $\mathit{coerce}_{\langle s_1', s_2\rangle, \langle s_1, s_2'\rangle} = \mathit{map}(s_1' \geq s_1, s_2 \leq s_2')$
- $\mathit{join}_{\langle s_1', s_2\rangle, \langle s_2', s_3\rangle, \langle s_1, s_3'\rangle} =$
$$\begin{array}{r} T(s_1', s_2) \circ T(s_2', s_3) \\ \mathit{map}(s_1' \geq s_1, s_2 \leq s_2) * \mathit{map}(s_2' \geq s_2, s_3 \leq s_3') \big\downarrow \\ T(s_1, s_2) \circ T(s_2, s_3') \\ \mathit{join}_{s_1, s_2, s_3'} \big\downarrow \\ T(s_1, s_3') \end{array}$$

The necessary equivalences hold due to the equational requirements of parameterized monads.

Given a productoid $\langle P, \mathit{map}, \mathit{unit}, \mathit{coerce}, \mathit{join}\rangle$ for $\mathcal{E}_{\langle S, \leq\rangle}$, define the parameterized monad for $\langle S, \leq\rangle$ as follows:

- $T(s, s') = \langle P_{\langle s, s'\rangle}, \mathit{map}_{\langle s, s'\rangle}\rangle$
- $\mathit{map}(s_1' \geq s_1, s_2 \leq s_2') = \mathit{coerce}_{\langle s_1', s_2\rangle, \langle s_1, s_2'\rangle}$
- $\mathit{unit}_s = \mathit{unit}_{\langle s, s\rangle}$
- $\mathit{join}_{s, s', s''} = \mathit{join}_{\langle s, s'\rangle, \langle s', s''\rangle, \langle s, s''\rangle}$

The necessary equivalences hold to due to the equational requirements of productoids.

These two processes are clearly inverses of each other. $\square$

Now that we have seen how parameterized monads fit within our framework, let us consider the expressiveness of parameterized monads. First, a parameterized monad is *not* a family of monads. While $T(s, s)$ forms a monad since the input and output are the same, this is not the case for $T(s, s')$ whenever s and s' are not equivalent to each other. This is because $\varepsilon \mapsto \langle s, s'\rangle$ holds only when s is a substate of s' and $\langle s, s'\rangle \fatsemi \langle s, s'\rangle \mapsto \langle s, s'\rangle$ holds only when s' is a substate of s. So, the effectoid $\mathcal{E}_{\langle S, \leq\rangle}$ is *neither* centric, idempotent, nor increasing in general. Thus the very notion of distinct states is inexpressible by indexed monads and layered monads, indicating how powerful parameterized monads are.

Parameterized monads have their limitations, though. For example, information-flow effect systems [29] and contextual effects [21] do not meet the requirements of Theorem 4 and so cannot be formalized using parameterized monads. Interestingly, though, should we allow EFF for $\mathcal{E}_{\langle S, \leq\rangle}$ to be just a subset of $S \times S$ rather than all pairs of states, then we can represent information-flow effect systems and contextual effects. For example, in [29] the states are levels of secrecy, so EFF contains only those pairs $\langle s, s'\rangle$ where s is a lower level of secrecy than s', since inputs can be propagated to outputs. Thus, Theorem 4 suggests such effect systems must be formalizable by something very similar to a parameterized monad.

Even such generalizations are still too restrictive for some effectoids. For example, the full effectoid for Crit *including* the basic effect ε cannot be formalized by a parameterized monad. The issue is that, ignoring substates since they are orthogonal to the following concerns, a parameterized monad can only have one effect between any two states. However, both ε and critical computations are permitted when the lock is acquired and finish with the lock acquired. If we wanted to prevent race conditions, then we would want to distinguish ε and critical computations since we should allow critical computations be ran in parallel with ε computations but not with other critical computations. Furthermore, if we wanted to track other effectful attributes that are state agnostic, such as non-determinism, then we would need multiple effects between any two states. Even just an indexed monad can be viewed as having many effects between a single state, showing why non-trivially indexed monads cannot be formalized by parameterized monads. Thus, while parameterized monads are expressive, there are still useful effectoids inexpressible even by generalized parameterized monads but still expressible by our framework.

Arrows and Freyd Categories Arrows [8], another generalization of monads, are a little difficult to discuss because they are simultaneously too general and too restrictive. First, they have three components, which we describe informally:

- a way to sequentially compose arrows
- a way to turn a pure computation into an arrow
- a way to propagate additional context through an arrow

These three components must satisfy equalities which essentially indicate that the arrows form a category, the pure computations form a pure subcategory, and a pure computation can be executed alongside an arrow without altering the overall effect of the arrow.

In fact, the above intuition behind the equational properties has been formalized. Heunan and Jacobs proved that arrows are equivalent to Freyd categories [10], an attempt to formalize arbitrary effectful computations. Informally, a Freyd category [26] is a premonoidal category [25] (a category with a notion of extending morphisms to propagate context) with a wide cartesian monoidal subcategory of pure morphisms that can be executed alongside other morphisms without altering their overall effect.

Note that an important aspect of the above descriptions is the notion of extending and propagating context. Herein lies our primary criticism of arrows and Freyd categories. What we determined is that nearly all of their structure has to do with extending and propagating context. If one removes the components that are present to serve those roles (i.e. anything using $\otimes$), what remains is just a category of effectful computations with a wide subcategory of pure computations. That is, arrows and Freyd categories say nothing about sequencing effectful computations besides the fact that sequencing pure computations results in a pure computation (not to mention they only handle one effect). So, while arrows generalize strong monads, they do so at the cost of discarding most of the useful structure to work with. It is for this reason that we focused on producer effects; our investigations suggest that being as general as possible means not providing any structure beyond a category with a distinguished subcategory.

8. Generality

The intent of this work is not to generalize prior semantic frameworks for effects, but to design the most general such framework possible. However, as we just mentioned while discussing arrows, the most general framework possible is not very useful. As such, we focused on producer effects and defined productors for formalizing producer effect systems. Here we show how a few language properties guarantee that the semantics of sequential composition forms a productor.

Figure 6 extends the language assumptions made in Figure 4. Again, while we assume these rules can be imposed upon the language at hand, we expect the language will have a different syntax and more rules in addition to those we assume.

Figure 6 introduces two new judgements. The judgement $\Gamma \vdash p \dashv \Gamma'$ indicates that p is a pure program with input Γ and output Γ' for some notion of purity. In prior work, this separation usually is termed as values versus computations. For example, OCaml restricts type generalization to values rather than arbitrary computations [39]. However, not all notions of purity may be restricted to values. For example, Haskell's notion of purity includes exceptions and non-termination. Here we let the designer determine their own notion of purity. Then we will show how effectful computations can be expressed by a productor on pure computations for whatever notion of purity the designer decided upon.

The other new judgement $\vdash p \overset{\varepsilon}{\underset{\Gamma\ \ \Gamma'}{=\!=}} p'$ indicates that p and p', viewed as ε-effectful computations from Γ to Γ', are semantically equivalent. When ε is absent, then they are semantically equivalent

$$\textit{Seq}\ \frac{\Gamma_0 \vdash p_1 \dashv \Gamma_1 \quad \ldots \quad \Gamma_{n-1} \vdash p_n \dashv \Gamma_n}{\Gamma_0 \vdash p_1; \ldots; p_n \dashv \Gamma_n} \qquad \textit{Seq}_{\not\varepsilon}\ \frac{\Gamma \vdash p \dashv \Gamma' \quad \Gamma' \vdash p' \dashv \Gamma'' \mid \varepsilon \quad \Gamma'' \vdash p'' \dashv \Gamma'''}{\Gamma \vdash p; p'; p'' \dashv \Gamma''' \mid \varepsilon}$$

$$\textit{Thunk}\ \frac{\Gamma \vdash p \dashv \Gamma' \mid \varepsilon}{\Gamma \vdash \lfloor p \rfloor_\varepsilon \dashv (\!|\Gamma'|\!)_\varepsilon} \qquad \textit{Exec}\ \frac{}{(\!|\Gamma|\!)_\varepsilon \vdash \mathbf{exec}_\varepsilon \dashv \Gamma \mid \varepsilon}$$

$$\textit{Eq}_\beta\ \frac{\Gamma \vdash p \dashv \Gamma' \mid \varepsilon}{\vdash \lfloor p \rfloor_\varepsilon; \mathbf{exec}_\varepsilon \overset{\varepsilon}{\underset{\Gamma\ \ \Gamma'}{=\!=}} p} \qquad \textit{Eq}_\eta\ \frac{\Gamma \vdash p \dashv (\!|\Gamma'|\!)_\varepsilon}{\vdash \lfloor p; \mathbf{exec}_\varepsilon \rfloor_\varepsilon \underset{\Gamma\ \ (\!|\Gamma'|\!)_\varepsilon}{=\!=} p}$$

Figure 6. Typing and semantic rules that guarantee productors

when viewed as pure computations. While not explicitly stated in Figure 6, we assume semantic equivalence is congruent with respect to sequential composition, implicit coercion of pure computation into effectful computation, and thunking.

Figure 6 also introduces three new meta-operations. The operation $\lfloor - \rfloor_\varepsilon$ maps an ε-effectful program to a pure thunked version of that program, delaying its execution and treating it as a value. In OCaml, given an ($\not\varepsilon$-effectful) expression e, $\lfloor e \rfloor_{\not\varepsilon}$ would be `fun () ->` e, effectively delaying the evaluation of e by turning it into a function waiting for an input, namely `()`. The operation $(\!|-|\!)_\varepsilon$ maps a context Γ to the context representing a thunked ε-effectful computation that would produce Γ whenever it is executed, i.e. the output of a thunked computation $\lfloor p \rfloor_\varepsilon$. In OCaml, $(\!|\tau|\!)_{\not\varepsilon}$ would be `unit ->` τ. Lastly, the operation $\mathbf{exec}_\varepsilon$ is the program that finally executes a thunked ε-effectful computation. In OCaml, $\mathbf{exec}_{\not\varepsilon}$ would be $-$ `()`, i.e. the program finally passing `()` to the thunked computation of the form `fun () ->` e. Note that, while our OCaml example defines the operations by wrapping the relevant program or context, other examples may be done by a more in-depth translation. For example, if a language has sum types, then thunking an exception-throwing program can be done by recursively translating the entire program to inject left or right instead of throwing an exception and to use pattern matching to explicitly propagate the exception. Thus a language does not have to be higher order in order to satisfy the requirements of Figure 6.

The rules in Figure 6 formalize a number of language properties, some of which are obvious, and some of which are fundamental to the notion of producer effects. *Seq* indicates that pure computations are closed under sequential composition. Note in particular that the empty computation is pure. $\textit{Seq}_{\not\varepsilon}$ indicates that sequential composition of effectful computations with pure computation preserves the effect of a computation. This formalizes the idea that the effect of a computation is not dependent on the values of its inputs or outputs. Note that these rules combined with using the empty computation for $\textit{Seq}_{\not\varepsilon}$ admit the following:

$$\frac{\Gamma \vdash p \dashv \Gamma' \qquad \not\varepsilon \mapsto \varepsilon}{\Gamma \vdash p \dashv \Gamma' \mid \varepsilon}$$

Thunk and *Exec* formally evidence the notion of producer effects. First, any effectful computation can be thunked into a pure computation by modifying only the output in a uniform manner. Second, there is a similarly effectful computation that executes a thunked computation. $\textit{Eq}_\beta$ and $\textit{Eq}_\eta$ essentially correspond to β- and η-equivalence for these constructs.

Note that these are not the only rules of the language; they are simply properties that may be superimposed upon a language. In particular, there can be (and typically are) many rules that operate on contexts of form $(\!|\Gamma|\!)_\varepsilon$ so that $\mathbf{exec}_\varepsilon$ is not necesarilly the only operation that can be performed on a thunked computation. Nonetheless, regardless of what additional rules the language at

hand may have, so long as it admits at least the rules in Figures 4 and 6 then it is guaranteed to be an instance of our framework.

With these we can finally present our fundamental theorem:

Theorem 5. If a language with an effector $\langle \textsc{Eff}, \overset{\fatsemi}{\mapsto} \rangle$ admits at least the rules in Figures 4 and 6, then there is a productor $\langle (\!|-|\!), map, join \rangle$ using pure computations modulo semantic equivalence as $\mathbf{Sem}$, such that the language admits the following:

$$\frac{\Gamma_0 \vdash p_1 \dashv \Gamma_1 \mid \varepsilon_1 \quad \dots \quad \Gamma_{n-1} \vdash p_n \dashv \Gamma_n \mid \varepsilon_n \qquad [\varepsilon_1, \dots, \varepsilon_n] \overset{\fatsemi}{\mapsto} \varepsilon}{\lfloor p_1; \dots; p_n \rfloor_\varepsilon \underset{\Gamma_0\, (\!|\Gamma_n|\!)_\varepsilon}{=\!=} \lfloor p_1 \rfloor_{\varepsilon_1}; map_{\varepsilon_1}(\dots(\lfloor p_n \rfloor_{\varepsilon_n}))\,; join_{[\varepsilon_1,\dots,\varepsilon_n],\varepsilon}}$$

Proof. Define the productor as follows:

- $P_\varepsilon(\Gamma)$ is already defined as $(\!|\Gamma|\!)_\varepsilon$
- $map_\varepsilon(p) = \lfloor \mathbf{exec}_\varepsilon; p \rfloor_\varepsilon$
- $join_{[\varepsilon_1,\dots,\varepsilon_n],\varepsilon} = \lfloor \mathbf{exec}_{\varepsilon_1}; \dots; \mathbf{exec}_{\varepsilon_n} \rfloor_\varepsilon$

In the technical report [33], using Eq_β, Eq_η, and congruence. Here we prove only the desired semantic property of sequential composition, using approximately the same proof strategy.

First, there is an important lemma: given pure programs p and p' from Γ to Γ', if $p; \mathbf{exec}_\varepsilon$ is semantically equivalent to $p'; \mathbf{exec}_\varepsilon$, then p and p' are semantically equivalent. The lemma assumptions and congruence imply that $\lfloor p; \mathbf{exec}_\varepsilon \rfloor_\varepsilon$ is semantically equivalent to $\lfloor p'; \mathbf{exec}_\varepsilon \rfloor_\varepsilon$. Eq_η then tells us that the former is semantically equivalent to p and the latter to p'. Thus, by transitivity, p and p' are semantically equivalent.

Now, to prove that $\lfloor p_1; \dots; p_n \rfloor_\varepsilon$ is semantically equivalent to $\lfloor p_1 \rfloor_{\varepsilon_1}; map_{\varepsilon_1}(\dots(\lfloor p_n \rfloor_{\varepsilon_n})); join_{[\varepsilon_1,\dots,\varepsilon_n],\varepsilon}$, due to our lemma we can instead prove $\lfloor p_1; \dots; p_n \rfloor_\varepsilon; \mathbf{exec}_\varepsilon$ is semantically equivalent to $\lfloor p_1 \rfloor_{\varepsilon_1}; map_{\varepsilon_1}(\dots(\lfloor p_n \rfloor_{\varepsilon_n})); join_{[\varepsilon_1,\dots,\varepsilon_n],\varepsilon}; \mathbf{exec}_\varepsilon$. So, $\lfloor p_1; \dots; p_n \rfloor_\varepsilon; \mathbf{exec}_\varepsilon$ is semantically equivalent to $p_1; \dots; p_n$ by Eq_β. From Eq_β and the definition of $join$, $join_{[\varepsilon_1,\dots,\varepsilon_n],\varepsilon}; \mathbf{exec}_\varepsilon$ is semantically equivalent to $\mathbf{exec}_{\varepsilon_1}; \dots; \mathbf{exec}_{\varepsilon_n}$. From Eq_β and the definition of map, $map_{\varepsilon_i}(p); \mathbf{exec}_{\varepsilon_i}$ is semantically equivalent to $\mathbf{exec}_{\varepsilon_i}; p$. Using that fact and Eq_β, we know that $\lfloor p_i \rfloor_{\varepsilon_i}; map_{\varepsilon_i}(\dots(\lfloor p_n \rfloor_{\varepsilon_n})); \mathbf{exec}_{\varepsilon_i}; \dots; \mathbf{exec}_{\varepsilon_n}$ is equivalent to $p_i; \lfloor p_{i+1} \rfloor_{\varepsilon_{i+1}}; map_{\varepsilon_{i+1}}(\dots(\lfloor p_n \rfloor_{\varepsilon_n})); \mathbf{exec}_{\varepsilon_{i+1}}; \dots; \mathbf{exec}_{\varepsilon_n}$. Via induction, $\lfloor p_1 \rfloor_{\varepsilon_1}; map_{\varepsilon_1}(\dots(\lfloor p_n \rfloor_{\varepsilon_n})); \mathbf{exec}_{\varepsilon_1}; \dots; \mathbf{exec}_{\varepsilon_n}$ is equivalent to $p_1; \dots; p_n$. Taking advantage of congruence, then $\lfloor p_1 \rfloor_{\varepsilon_1}; map_{\varepsilon_1}(\dots(\lfloor p_n \rfloor_{\varepsilon_n})); join_{[\varepsilon_1,\dots,\varepsilon_n],\varepsilon}$ is also equivalent to $p_1; \dots; p_n$. Therefore the desired semantic property holds. $\square$

Example Our toy language Crit does not actually satisfy the requirements of Figure 6. This illustrates that Figure 6 is sufficient but not necessary for our framework, demonstrating that our framework is useful even for formalizing languages without a notion of thunks. Nonetheless, for sake of illustration, we show how we might extend Crit so that it models Figure 6.

A sufficient extension of Crit is shown in Figure 7. Semantically we assume the usual β- and η-equivalences. The first extension is a singleton type Unit. The second and more important extension is effectful function types. In particular, $\tau \xrightarrow{\varepsilon} \Gamma$ represents an ε-effectful function accepting a τ as input and during execution assigning values of the appropriate types to the variables in Γ. Thus this extension is like the computational lambda calculus with multiple effects mixed with a register-based language.

With this language we can define the required judgements and operations in Figure 6 as follows:

$$\Gamma \vdash p \dashv \Gamma' \equiv \Gamma \vdash p \dashv \Gamma' \mid \mathbb{\varepsilon} \qquad (\!|\Gamma|\!)_\varepsilon \equiv t : \mathsf{Unit} \xrightarrow{\varepsilon} \Gamma$$
$$\lfloor p \rfloor_\varepsilon \equiv t := (\lambda u. \langle\rangle := u; p) \qquad \mathbf{exec}_\varepsilon \equiv u := \langle\rangle; t\, u$$

That is, a pure computation is one with the basic effect $\mathbb{\varepsilon}$. Thunking works by turning the computation into a function waiting for a

$$\frac{}{\varnothing \vdash u := \langle\rangle \dashv u : \mathsf{Unit} \mid \mathbb{\varepsilon}} \qquad \frac{\Gamma, x : \tau \vdash p \dashv \Gamma' \mid \varepsilon}{\Gamma \vdash f := \lambda x.p \dashv f : (\tau \xrightarrow{\varepsilon} \Gamma') \mid \mathbb{\varepsilon}}$$

$$\frac{}{u : \mathsf{Unit} \vdash \langle\rangle := u \dashv \varnothing \mid \mathbb{\varepsilon}} \qquad \frac{}{f : (\tau \xrightarrow{\varepsilon} \Gamma), x : \tau \dashv f\, x \vdash \Gamma \mid \varepsilon}$$

Figure 7. Crit extended with effectful functions

Unit value, eliminating that unit value, and then finally running the computation. That function is stored into variable t so that thunking results in an output context of the form $t : \mathsf{Unit} \xrightarrow{\varepsilon} \Gamma$. Executing a thunked computation, then, simply involves storing the unique value of Unit into a variable and then calling the thunk with that variable as the argument, thus causing the body of the function to finally execute.

These definitions clearly satisfy the requirements of Figure 6. Thus, our fundamental theorem implies that the semantics of extended Crit can be formalized with a productor defined on just the extended Crit computations with the basic effect $\mathbb{\varepsilon}$. Furthermore, the theorem shows how to construct this productor.

Closed Freyd Categories Now we would like to revisit existing frameworks to see what insights the assumptions in Figure 6 can offer. As we discussed, while Freyd categories offer a lot of useful structure for working with contexts, they provide little structure regarding sequential composition. However, Power and Thielecke recognized that strong monads arise for *closed* Freyd categories [27]. This is no surprise given our theorem, since the structures that makes a Freyd category closed are essentially the assumptions in Figure 6. Similarly, Atkey defined a notion of *parameterized* Freyd categories [2], and observed that closed parameterized Freyd categories gave rise to parameterized monads. Again, the additional structure for a closed parameterized Freyd category corresponds to the assumptions in Figure 6, so this result is no surprise given our theorem.

9. Conclusion

We have presented productors, a semantic framework for sequential composition of computations with producer effects, a concept we were able to formalize abstractly. In particular, we showed why the widespread notion of thunking makes producer effects so common. In our discussions of existing frameworks, we argued why it is important to restrict our attention to producer effects. We illustrated how monadic frameworks fit within productors, and used our framework to illustrate how properties of the effect system at hand give rise to various semantic structures.

In all this discussion, we have elided the higher categorical connections. For example, productors can be viewed as functors from the effector to the 2-category of categories. This higher perspective suggests some ways to adapt the framework to more specialized forms of computation. For example, by changing the target 2-category to that of premonoidal categories (i.e. categories with a notion of propagating context), one arrives at productors that can propagate context, such as a strong monad. Or, by using the 2-category of categories and *partial* functors, one can drop the implicit assumption of Theorem 5 that $(\!|\Gamma|\!)_\varepsilon$ is defined for all contexts Γ even if there is no ε-effectful computation with Γ as its output. In another direction, while we chose to present effectors along the line of lax List-monad algebras in $\mathbf{Rel}$, they can equivalently be described as thin multicategories [15]. It would be interesting to investigate how the concepts for multicategories translate to effectors and productors. For example, a representable thin multicategory is equivalently a total principalled effectoid (equivalent to a congruently preordered monoid).

Our preliminary investigations into the dual concept of consumer effects and consumptors have been very intriguing. The traditional notion of context seems to be well described as a consumptor. Strength appears to be formalizable as an interplay of this consumer effect with the producer effects. Similarly, non-linear uses of inputs also seem best described as a consumptor. After all, an intuitionistic implication $P \Rightarrow Q$ translates to linear implication $!P \multimap Q$ with a modification on the *input* rather than the output. The ! modality is a comonad, which is a special class of consumptors just like monads are a special class of productor. Furthermore, it seems that strictness and laziness arise as two dual ways to make such a consumptor interact with producer effects.

Finally, while we have discussed what the semantic framework for producer effects should look like, we have not investigated how to actually build productors. With monads, there have been a variety of techniques for composing monads [11] (though interestingly many of these, such as distributive laws [14], arise as special cases of our framework), building monads from monad transformers [16], or combining algebraic monads [24] with tensors and sums [9]. We would like to see how these concepts extend to productors. For example, the coproduct of two monads is relatively simple to define [18], but at first thought it is not clear whether the coproduct of two effectors and respective productors would be simpler or more complex. Also, transformers such as for exceptions rely heavily on the presence of a $unit$ operation, so one wonders if they are restricted to just centric effectors. By reexamining these techniques in this new light, we expect to acquire a better understanding of their fundamental structure.

With this framework we aim to lay new grounds for the foundations of programming languages. Of course, there are still many more forms of composition to formalize the semantics of, but we have already begun transferring the strategy taken here to those settings and found some fruitful results. In the end we expect to have composable frameworks for formalizing the many roles effects have in programming languages. We hope this work will provide a new means to abstractly formalize, expand, and communicate programming languages.

Acknowledgements In addition to our anonymous reviewers, we thank John C. Baez, Thomas Ball, Daniel Brown, Jeffrey S. Foster, Michael Hicks, Ohad Kammar, Daan Leijen, Sorin Lerner, Daniel Marino, Andrew Myers, Patrick Rondon, Michael Shulman, and Zachary Tatlock for their valuable feedback on the research, its context, and our writing.

References

[1] Martín Abadi. Access control in a core calculus of dependency. In *ICFP*, 2006.

[2] Robert Atkey. Parameterised notions of computation. *JFP*, 19:335–376, July 2009.

[3] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In *International Summer School on Applied Semantics*, 2000.

[4] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical report, University of Copenhagen, 1989.

[5] Andrzej Filinski. Representing layered monads. In *POPL*, 1999.

[6] Andrzej Filinski. Monads in action. In *POPL*, 2010.

[7] Roger Godement. *Topologie Algébrique et Théorie des Faisceaux*. Hermann, 1958.

[8] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, May 2000.

[9] Martin Hyland, Gordon Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1):70–99, 2006.

[10] Bart Jacobs, Chris Heunen, and Ichiro Hasuo. Categorical semantics for arrows. *JFP*, 19:403–438, 2009.

[11] Mark P. Jones and Luc Duponcheel. Composing monads. Technical report, Yale University, New Haven, CT, USA, December 1993.

[12] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL*, 1993.

[13] Richard B. Kieburtz. Taming effects with monadic typing. In *ICFP*, 1998.

[14] David J. King and Philip Wadler. Combining monads. In *ETAPS*, 1992.

[15] Tom Leinster. *Higher Operads, Higher Categories*. Cambridge University Press, 2004.

[16] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL*, 1995.

[17] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *POPL*, 1988.

[18] Christoph Lüth and Neil Ghani. Composing monads using coproducts. *ACM SIGPLAN Notices*, 37(9):133–144, 2002.

[19] Daniel Marino and Todd Millstein. A generic type-and-effect system. In *TLDI*, 2009.

[20] Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, 1989.

[21] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *POPL*, 2008.

[22] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *ACM Computing Surveys*, 1999.

[23] Hanne Riis Nielson, Flemming Nielson, and Torben Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In *LOMAPS*, 1997.

[24] Gordon Plotkin and John Power. Notions of computation determine monads. In *FoSSaCS*, 2002.

[25] John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7:453–468, October 1997.

[26] John Power and Hayo Thielecke. Environments, continuation semantics and indexed categories. In *TACS*, 1997.

[27] John Power and Hayo Thielecke. Closed Freyd- and κ-categories. In *ICAL*, 1999.

[28] John C. Reynolds. Using category theory to design implicit conversions and generic operators. *LNCS*, 94:211–258, 1980.

[29] Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. Lightweight monadic programming in ML. Technical report, Microsoft Research, 2011.

[30] Jean-Pierre Talpin. *Theoretical and Practical Aspects of Type and Effect Inference*. PhD thesis, École des Mines de Paris and University Paris VI, Paris, France, 1993.

[31] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *JFP*, 2:245–271, 1992.

[32] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[33] Ross Tate. The sequential semantics of producer effect systems. Technical report, Cornell University, 2012.

[34] Andrew P. Tolmach. Optimizing ML using a hierarchy of monadic types. In *Types in Compilation*, 1998.

[35] Philip Wadler. Comprehending monads. In *LISP and Functional Programming*, 1990.

[36] Philip Wadler. The essence of functional programming. In *POPL*, 1992.

[37] Philip Wadler. Monads and composable continuations. *LISP and Symbolic Computation*, 7:39–56, January 1994.

[38] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *Transactions on Computational Logic*, 4(1):1–32, 2003.

[39] Andrew K. Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8(4):343–355, 1995.