A Flexible Semantic Framework for Effects

Ross Tate
University of California, San Diego
rtate@cs.ucsd.edu

Daan Leijen
Microsoft Research, Redmond
daan@microsoft.com

Sorin Lerner
University of California, San Diego
lerner@cs.ucsd.edu

Abstract

Effects are a powerful and convenient component of programming. They enable programmers to interact with the user, take advantage of efficient stateful memory, throw exceptions, and nondeterministically execute programs in parallel. However, they also complicate every aspect of reasoning about a program or language, and as a result it is crucially important to have a good understanding of what effects are and how they work. In this paper we present a new framework for formalizing the semantics of effects that is more general and thorough than previous techniques while clarifying many of the important concepts. By returning to the category-theoretic roots of monads, our framework is rich enough to describe the semantics of effects for a large class of languages including common imperative and functional languages. It is also capable of capturing more expressive, precise, and practical effect systems than previous approaches. Finally, our framework enables one to reason about effects in a language-independent manner, and so can be applied to many stages of language design and implementation in order to create more broadly applicable tools for programming languages.

1. Introduction

Like mathematical functions, program procedures take inputs and produce outputs. Unlike mathematical functions, program procedures may also read from and write to stateful memory, interact with a user or the outside world in general, throw exceptions, fail to terminate, or terminate with a non-deterministic result. We call all these differences *effects*. Effects are an integral component of programming languages. For one, they provide the programmer convenient access to the powerful and efficient capabilities of the machine such as interrupts, stateful memory, the file system, and the monitor. Even supposedly pure languages such as Haskell use implicit non-termination, and Haskell even provides a means for users to define and use their own effects via monads and imperative functional programming [9]. Other languages such as in parsing tools use implicit enumerable non-determinism so that programmers may specify search problems in a concise manner.

Since effects are so integral to programming languages, it is important to have a formalization of their usage and semantics. This formalization should be *thorough*, addressing all ways in which effects impact the semantics of a language. Furthermore, this formalization should be *general*, making only minimal assumptions so that it can be applied to as many languages as possible.

Here we present a framework for formalizing the denotational semantics of effects. Our framework is thorough, identifying many of the roles that effects play in languages, some of which have not been addressed before. For each role we present both a *nominal* component for *naming* effects (this is the space in which type-and-effect analyses [15, 17, 22, 23, 24] work) and a *semantic* component for defining the *denotational semantics* of effects (this is the space in which monadic techniques typically work). Our framework is also general, making only the assumptions necessary for semantics to be *coherent*, which informally means that all sensible ways to insert implicit semantic operators lead to equivalent semantics. Examples of coherence requirements are Reynolds' requirements for implicit coercions [20] and the monad laws for imperative functional programming [9].

Our framework is both more general and more thorough than the predominant technique for formalizing the semantics of effects, namely monadic techniques [2, 9, 18, 19, 25]. As evidence of our framework's generality, in this paper we provide two effect systems whose semantics our framework can formalize but which cannot be formalized using monadic techniques, contrary to the claim made by Wadler and Thiemann that the semantics of any effect system can be formalized using a hierarchy of monads and monad morphisms [27]. Our framework is more general because it emphasizes the *interaction* of effects whereas monadic techniques typically treat each effect individually. Using our framework, we are able to classify and prove precisely which effect systems satisfy Wadler and Thiemann's claim, and what kinds of interactions monadic techniques are able to formalize.

Furthermore, our framework is more thorough than monadic techniques, addressing many roles of effects which cannot be properly formalized using monads. We introduce *lateral composition* for combining effectful arguments; monads are only able to formalize left-to-right or right-to-left evaluation, but not all languages use one semantics or the other. We introduce *flexible* semantics for effects meant to give the compiler some freedom of choice; monads are not capable of formalizing semantics which, for example, allow the compiler to choose which order to evaluate effectful arguments. We introduce *infinite* effects for formalizing the denotational semantics of effects in unbounded loops and recursion; Wadler and Thiemann rely on operational small-step semantics for formalizing non-terminating processes [27], and Haskell implicitly relies on lazy data structures. Thus, although monads were a key inspiration, our framework is much more thorough and general in its treatment of effects.

Our framework is also useful beyond formalizing semantics. To demonstrate this, we apply our framework to three problems across the spectrum of language design and implementation, providing language-independent solutions to each problem. First we show that the value restriction [28] can be relaxed in the presence of effects satisfying an abstract property phrased in terms of our framework. Second we define abstract properties of effects in the presence of which common-subexpression elimination and dead-code elimination can be applied. Third we specify the abstract properties of an effect which allow computations in separate threads to be interwoven arbitrarily per the requirements of parallelization, significantly relaxing the requirements of commutative monads [8]. Since each solution specifies its requirements in terms of our abstract framework for effects, these solutions can be applied to any language in any circumstance satisfying those abstract requirements.

We hope our framework will help unify programming languages research by enabling future tools to specify their requirements abstractly rather than be confined to a specific language with unknown implicit requirements. Following our own philosophy, we state the assumptions we make in this paper: we assume that we are working with strict programming languages without linear or dependent types. We make these assumptions not because of any fundamental limitations of our framework, but because otherwise we would need significantly more complex abstractions such as fibred-category theory [5]. Although the applications of our work to these domains are also interesting, we feel that they would distract from the main points of the paper and so here we focus on strict programming languages with classical type systems.

To summarize, this paper makes the following contributions:

Section 2 A definition of nominal effect systems for naming effects, such as type-and-effect systems used in analyses

Section 3 A semantic framework for formalizing the denotational semantics of nominal effect systems

Section 4 A classification of the nominal effect systems whose semantics can be formalized by monads and monad morphisms, and examples of systems for which this is not possible

Section 5 Extensions for effect systems with lateral composition, flexible semantics, and infinite effects

Section 6 Applications of our framework to type generalization, program optimization, and parallelism

In Section 7, we conclude the paper with a summary of semantic effect systems and insights into future extensions of this work.

2. Nominal Effect Systems

Effects have become a somewhat overloaded concept. For some people, effects are like types, and can be used in a programming language's type system or in an optimization's analytical framework. For others, effects are a semantic impurity of procedures. These two perspectives are indeed related in that the former essentially *names* effects while the latter *gives meaning* to effects. In this section we present *nominal* effect systems; that is systems for *naming* effects. In the section afterwards, we present *semantic* effect systems; that is systems which give semantics to nominal effect systems. We make this distinction both for the sake of clarity and because it can be convenient to reason about these two systems separately.

In this section and the next, we focus on nominal and semantic effect systems for only *sequential* programs of the form

$$\begin{array}{l} x \leftarrow e; \\ y \leftarrow e'; \\ \texttt{return}\ e'' \end{array}$$

In Section 5, we will show how to add more components to the systems we present here in order to formalize effect systems for more realistic programs in which not only each line of code has an effect but also subexpressions have effects, and for programs with infinite processes such as unbounded loops and recursion. But for now we focus on just sequential programs because these are what existing systems for formalizing semantics of effects have focused on. In Section 4, we will show how these existing systems fit within our framework, and how our framework is more general.

2.1 Nominal Effects, Subeffects, and Sequential Composition

Procedures, unlike mathematical functions, are effectful, and a nominal effect system is a classification of these effects. There are already many nominal effect systems in existence. Most of these are static systems, such as the type-and-effect systems used by optimizing compilers, or the instances of the `Monad` type class in Haskell. However, in the same way that there are dynamic counterparts to static types, there can also be dynamic nominal effect systems particularly useful for defining the semantics of a language.

$$\textsc{Eff} = \{\mathbb{c}, \mathsf{read}(S), \mathsf{set}(S), \mathsf{update}(S), \mathsf{state}(S)\}$$

$$\begin{array}{ccc} & \mathbb{c} \leq \mathsf{read}(S) & \\ & \wedge \qquad \wedge & \\ \mathsf{set}(S) \leq \mathsf{update}(S) & & \leq \mathsf{state}(S) \end{array}$$

$\downarrow \fatsemi \rightarrow$	$\mathsf{read}(S)$	$\mathsf{set}(S)$	$\mathsf{update}(S)$	$\mathsf{state}(S)$
$\mathsf{read}(S)$	$\mathsf{read}(S)$	$\mathsf{state}(S)$	$\mathsf{state}(S)$	$\mathsf{state}(S)$
$\mathsf{set}(S)$	$\mathsf{set}(S)$	$\mathsf{set}(S)$	$\mathsf{set}(S)$	$\mathsf{set}(S)$
$\mathsf{update}(S)$	$\mathsf{state}(S)$	$\mathsf{set}(S)$	$\mathsf{update}(S)$	$\mathsf{state}(S)$
$\mathsf{state}(S)$	$\mathsf{state}(S)$	$\mathsf{state}(S)$	$\mathsf{state}(S)$	$\mathsf{state}(S)$

Figure 1. Nominal Effect System for State Machines

Nominal effect systems, like type systems, often have some notion of subeffects. A procedure with nominal effect ε also has nominal effect ε' whenever ε is a subeffect of ε'. And, as with types, a subeffect is typically more precise or more restrictive than a supereffect. Thus nominal effect systems are essentially (static or dynamic) type systems for procedures.

Sequential nominal effect systems have a modular method for naming the effect of a sequence of processes. That is, if $line_1$ has effect ε_1 and $line_2$ has effect ε_2, then the effect of the sequence $\{line_1; line_2\}$ is some function of ε_1 and ε_2. For example, if the first line reads from the heap and the second line writes to the heap, then the combination reads-and-writes the heap.

And so we formalize nominal effect systems with subtyping and sequential composition using the following definition.

Definition. A nominal effect system is a set Eff of nominal effects (i.e. effect names) and a distinguished basic effect $\mathbb{c}$ given non-exclusively to effectless processes. A nominal subeffect system additionally has a preorder (reflexive transitive binary relation) $\leq$ on the set Eff. Alternatively, a sequential nominal effect system additionally has an associative binary operator $\fatsemi$ with $\mathbb{c}$ as the identity element, such that if a procedure has effect ε_1 and another procedure has effect ε_2, then the composition of those procedures has effect $\varepsilon_1 \fatsemi \varepsilon_2$. Should a nominal effect system have both subeffects and sequential composition, then sequential composition must preserve subeffects: $\varepsilon_1 \leq \varepsilon_1' \wedge \varepsilon_2 \leq \varepsilon_2' \implies \varepsilon_1 \fatsemi \varepsilon_2 \leq \varepsilon_1' \fatsemi \varepsilon_2'$. ◆

2.2 Example Nominal Effect Systems

Analytical effect systems are nominal effect systems with a static analysis that examines a program and determines the nominal effect of each operation in the program. These are often used by compilers in order to check whether certain optimizations would be sound, or by a verifier in order to understand the implicit operations in the program. They typically follow Talpin and Jouvelot's type-and-effect discipline [23] and fall within Marino and Millstein's generic type-and-effect system [17]. Due to their analytical nature, the subeffect system forms a join semi-lattice and sequential composition $\fatsemi$ is simply the join operator $\sqcup$ on this semi-lattice [10, 15, 24]. In particular, there is usually a set of primitive effects, and Eff is simply all finite sets of primitive effects [2, 17, 22, 23]. The basic effect $\mathbb{c}$ is the bottom of this join semi-lattice (e.g. the empty set) and embodies all effects which are not being tracked by the analytical effect system. In Section 4 we will see that the unification of the subeffect structure and the sequential composition structure has a significant impact on the denotational semantics of these effect systems. However, this unification does not hold for all nominal effect systems, as we will demonstrate shortly.

$$\begin{aligned} \text{EFF} &= \{\mathsf{nd}(n) \mid n \in \mathbb{N} \wedge n \geq 1\} \\ \mathbb{c} &= \mathsf{nd}(1) \\ \mathsf{nd}(m) \leq \mathsf{nd}(n) &= m \leq n \\ \mathsf{nd}(m) \fatsemi \mathsf{nd}(n) &= \mathsf{nd}(m * n) \end{aligned}$$

Figure 2. Nominal Effect System for Bounded Non-Determinism

A *typed effect system* is a nominal effect system which is actually integrated into the programming language's type system. Because of this integration, typed effect systems have the special privilege of having sequential composition to be a *partial* operation, rejecting programs whenever $\fatsemi$ is not defined on the effects of two lines of code. In particular, in Haskell where each non-basic effect is a `Monad` instance, $\varepsilon \fatsemi \varepsilon'$ is defined only when ε equals ε', meaning only lines belonging to the exact same `Monad` instance can be used together. Even $\varepsilon \fatsemi \mathbb{c}$ is undefined so that any effectless value must be *explicitly* coerced to an effectful value via the `return` operator. Having sequential composition be partial allows an interesting degree of flexibility in designing a nominal effect system, but for sake of simplicity we will treat sequential composition as total.

Another example of a nominal effect system is shown in Figure 1. This system captures the effects of a language for state machines, where S is the set of states of the machine. The basic effect $\mathbb{c}$ is for procedures which neither depend on nor change the current state. The effect $\mathsf{read}(S)$ is for procedures which depend on the current state. The effect $\mathsf{set}(S)$ is for procedures which set the current state. The effect $\mathsf{update}(S)$ is for procedures which update the current state. The effect $\mathsf{state}(S)$ is for procedures which depend on and update the current state. Intuitively, the subtle difference between the $\mathsf{set}(S)$ and $\mathsf{update}(S)$ effects is that $\mathsf{set}(S)$ procedures replace the *entire* state, whereas $\mathsf{update}(S)$ procedures only change *part* of the state. The primitive effectful operations for using and changing the state are the following:

$$\begin{aligned} \mathit{get} &: 1 \xrightarrow{\mathsf{read}(S)} S && \text{gets the current state} \\ \mathit{put} &: S \xrightarrow{\mathsf{set}(S)} 1 && \text{sets the current state} \\ \mathit{modify} &: (S \xrightarrow{\mathbb{c}} S) \xrightarrow{\mathsf{update}(S)} 1 && \text{updates the current state} \end{aligned}$$

We use 1 to represent the singleton type (i.e. unit). These three operations are how the $\mathsf{read}(S)$, $\mathsf{set}(S)$, and $\mathsf{update}(S)$ effects get introduced. The $\mathsf{state}(S)$ effect only results from these effects interacting, such as from a $\mathsf{set}(S)$ procedure occurring *after* a $\mathsf{read}(S)$ procedure. Note that a $\mathsf{read}(S)$ procedure occurring after a $\mathsf{set}(S)$ procedure, on the other hand, results in a $\mathsf{set}(S)$ procedure rather than a $\mathsf{state}(S)$ procedure. This may seem odd, and shortly we will formalize why, but in the next section we will use denotational semantics to prove that it is sound. We should also note that Haskell uses the same get, put, and modify functions in their monadic framework, but because of the coarseness of monads all these functions are given the same effect $\mathsf{state}(S)$, which is a significant loss of precision.

In Figure 2 we present a nominal effect system for bounding non-determinism; that is, a non-deterministic language in which the degree of non-determinism is tracked by the effect system. The effect $\mathsf{nd}(n)$ indicates that the procedure non-deterministically results in up to n different values. Primitive operations for a language with such an effect system would be the following:

$$\mathit{arb}_n : \tau^n \xrightarrow{\mathsf{nd}(n)} \tau$$

These operations non-deterministically select one argument to return. Sequential composition is particularly interesting because, although the subeffect system forms a lattice, sequential composition does not coincide with the join operation on this lattice as it did with analytical effect systems. To demonstrate why, consider the following program:

$$\begin{array}{l} x \leftarrow \mathit{arb}_2(1, 5); \\ y \leftarrow \mathit{arb}_2(x + 1, x - 1); \\ \texttt{return } y * 2 \end{array}$$

Even though each individual line non-deterministically selects between 2 values ($\mathsf{nd}(2)$), the combined result can return 4 different values ($\mathsf{nd}(4)$), whereas the join of $\mathsf{nd}(2)$ and $\mathsf{nd}(2)$ is $\mathsf{nd}(2)$ instead. This pattern is typical of effect systems which attempt to track or bound the imprecision in a program. A more practical such effect system would monitor the imprecision due to floating-point operations, but we present the bounded non-determinism effect system instead because its semantics is much simpler to formalize.

2.3 Classifying Nominal Effect Systems

One goal of our framework is to enable other tools to phrase their assumptions in terms of abstract properties of effects and effect systems. We have just presented a number of nominal effect systems each of which with very different properties. Here we identify some of the more significant properties that characterize the behavior of these systems. In the next section we will illustrate how these properties also impact the semantics of these systems, but for now we focus on the nominal level. In particular, the triple $\langle \text{EFF}, \mathbb{c}, \fatsemi \rangle$ forms a mathematical structure known as a monoid, and $\leq$ forms a congruence relation for that monoid, so we can apply concepts from monoid theory to classify nominal effect systems.

Insertable Effects A nominal effect ε is *insertable* if it is a supereffect of the basic effect $\mathbb{c}$. Recall that $\mathbb{c}$ is the effect given to effectless procedures, and so an effectless procedure can also be viewed as having any insertable effect ε due to the nature of supereffects. Since the identity function is effectless and can be inserted anywhere in a program without changing its semantics, any insertable effect can likewise be inserted anywhere in the program (although at a loss of precision). All the effects presented above are insertable (although Haskell requires an explicit `return` operation) except for one: the $\mathsf{set}(S)$ effect. An effectless procedure cannot be made into one which sets the state, at least not without significantly impacting the semantics of the program. Another more practical example of an uninsertable effect is the memory initialization effect; it is important to know whether memory has been initialized and a procedure which does not initialize memory cannot be safely treated as one which does. Thus it is important to acknowledge the effects which are uninsertable and explicitly state when one assumes the effects being used are insertable.

Sequentially Compressive Effects A nominal effect ε is *sequentially compressive* if $\varepsilon \fatsemi \varepsilon$ equals ε; in monoid theory this is known as an idempotent element. This means that the procedures with effect ε are closed under composition, and so there is a subspace of ε procedures. In analytical effect systems, Haskell's effect system, and the state machine effect system, *all* effects are sequentially compressive; these are known as idempotent monoids. However, the bounded non-determinism effect system has *no* sequentially compressive effects (except for the basic effect $\mathbb{c}$ which is always sequentially compressive). Not only are sequentially compressive effects convenient because they are closed under composition, but they form a fixed point and so are useful for formalizing loops and recursion, although in Section 6 we will see that additional structure is necessary as well.

Order-Insensitivity A sequential nominal effect system is *sequentially order-insensitive* if $\varepsilon \fatsemi \varepsilon'$ always equals $\varepsilon' \fatsemi \varepsilon$; in monoid theory this is known as commutativity. This means that the nominal effect of a sequence of lines does not depend on the order of those lines. All the nominal effect systems presented above are

$$
\begin{aligned}
T_{\mathsf{nd}(n)}(\tau) &= \{S \subseteq \tau \mid 1 \leq |S| \leq n\} \\
map_{\mathsf{nd}(n)}(f) &= \lambda S.\{f(x) \mid x \in S\} \\
unit(x) &= \{x\} \\
convert_{\mathsf{nd}(m) \leq \mathsf{nd}(n)}(S) &= S \\
join_{\mathsf{nd}(m),\mathsf{nd}(n)}(\mathcal{S}) &= \textstyle\bigcup_{S \in \mathcal{S}} S
\end{aligned}
$$

Figure 3. Semantic Effect System for Bounded Non-Determinism

order-insensitive except for the state machine effect system. In particular, the $\mathsf{set}(S)$ effect has the property that $\mathsf{set}(S) \fatsemi \varepsilon$ always equals $\mathsf{set}(S)$ (known as a left zero), but the reverse is not true. This captures the significance of order in a state machine. Interestingly, analytical effect systems cannot capture this significance because they are order-insensitive, since joins are always commutative.

Increasing A sequential nominal subeffect system is *sequentially increasing* if ε and ε' are always subeffects of $\varepsilon \fatsemi \varepsilon'$. This means that when we sequence two effects they are always contained within the combined result. Again this holds for all the nominal effect systems presented above except for the state machine effect system. This is what is so peculiar about the fact that $\mathsf{set}(S)$ followed by $\mathsf{read}(S)$ is $\mathsf{set}(S)$, which is not a supereffect of $\mathsf{read}(S)$, rather than $\mathsf{state}(S)$, which is a supereffect both. However, we believe that *not* requiring sequential composition to be increasing is one of the most powerful aspects of our framework because it enables effects to interact in a way *besides* containment. Nonetheless, many uses of effect systems implicitly assume that the nominal effect system is increasing.

3. Semantic Effect Systems

We have provided a number of nominal effect systems and interesting ways to classify nominal effect systems. However, it is important to realize that these are just naming schemes for effects and that they provide no evidence that the names actually *mean* what we expect them to mean. We could have just as easily specified get as having the $\mathsf{set}(S)$ effect, put as having the $\mathsf{read}(S)$ effect, and $\mathsf{read}(S) \fatsemi \mathsf{read}(S)$ as being $\mathbb{c}$. In this section we formalize *semantic effect systems* for nominal effect systems. Semantic effect systems give a denotational semantics to nominal effect systems, demonstrating that they act the way we think they should and that they are sound abstractions of the underlying effects.

Here we present sequential semantic effect systems which are a generalization of *monads*, the predominant technique for formalizing the semantics of effects. In 1958, Godement invented standard constructions [4], 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 [18]. In 1990, Wadler carried this concept over to the functional languages community [25, 26], and in 1993 these concepts were realized as monadic programming and added to Haskell to make I/O more convenient and to incorporate *imperative functional programming* [9]. Now, we show how to generalize monads capable of formalizing sequential programs with a *single* effect, to sequential semantic effect systems capable of formalizing multiple interacting effects.

3.1 The Semantics of Effects

Here we give semantics to nominal effect systems using concepts from category theory. We present the various components of sequential semantic effect systems through two running examples. The first example uses a very simple nominal effect system with only one effect: $\mathbb{c} = \mathsf{partial}$. The second example, shown in Figure 3, is the semantic effect system for the bounded non-determinism nominal effect system from Figure 2. Because the first example has only one effect, its semantics using our framework is formalized by a traditional monad. However, using the second example we will also show how to generalize to arbitrarily complex nominal effect systems. Thus our sequential semantic effect systems essentially generalize monads from one effect to multiple effects.

Effects as Functors Consider the expression $(64 \div x) + 1$ (using integer division). Forget that we all know what this expression means due to our years of experience with advanced arithmetic, and instead 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 $\mathsf{partial}$ effect. We might represent this by saying $\div$ has type $\mathbb{Z} \times \mathbb{Z} \xrightarrow{\mathsf{partial}} \mathbb{Z}$. We want to formalize what it means to have the $\mathsf{partial}$ effect. The observation made by Moggi [18] is that we can do this by modifying the return type of $\div$. In particular, we can view $\div$ as a function which returns an integer *or* a failure code. We can define an algebraic data type Partial to represent these two cases:

$$\text{Partial}(\tau) = \text{success}(\tau) \mid \text{failure}$$

Then $\div$ can be given the type $\mathbb{Z} \times \mathbb{Z} \to \text{Partial}(\mathbb{Z})$.

The next step is defining what to do should a failure occur. In particular, should $64 \div x$ fail, one expects $(64 \div x) + 1$ to fail as well. Thus, $(64 \div x) + 1$ also has the $\mathsf{partial}$ effect. In a sense, what we need to define is how an effect should be propagated through computations, specifically the computation $\lambda d.d + 1$. We can do this by using a map operation:

$$
\begin{aligned}
&map_{\mathsf{partial}} : (\tau \to \tau') \to (\text{Partial}(\tau) \to \text{Partial}(\tau')) \\
&map_{\mathsf{partial}}(f) = \lambda px.\ \mathbf{case}\ px \begin{cases} \text{success}(x) \mapsto \text{success}(f(x)) \\ \text{failure} \qquad\ \mapsto \text{failure} \end{cases}
\end{aligned}
$$

Thus map turns a normal computation into one which takes an effectful argument and propagates the effect. In this case, map_{partial} indicates that if a failure condition is present then all computation should be bypassed and the failure propagated. Using this function, we can formalize the semantics of $(64 \div x) + 1$ as $map_{\mathsf{partial}}(\lambda d.d + 1)(64 \div x)$. 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 will the entire expression.

This pair of a type constructor T : TYPE $\to$ TYPE and a function on computations $map : (\tau \to \tau') \to (T(\tau) \to T(\tau'))$ is called a *functor* (on the category of types), provided it satisfies a few additional equalities which we do not repeat here. In the setting of effects, the type constructor T indicates how the effect can be described as data, and the function on computations map defines how to propagate the effect through normal computations.

Definition. A semantic effect system for a nominal effect system $\langle \text{EFF}, \mathbb{c} \rangle$ specifies for each effect ε in EFF a functor $\langle T_\varepsilon, map_\varepsilon \rangle$ indicating how to describe the effect and propagate it through computations, and also specifies an operation $unit : \tau \to T_{\mathbb{c}}(\tau)$ (specifically a natural transformation) indicating how to give constants a trivial form of the basic effect. ◆

In the above definition we introduced one more operation: $unit$. The $unit$ operation specifies how to bring effectless values into this effectful world of computations by giving them the basic effect $\mathbb{c}$. This operation is related to monadic units, but it is required *only* for the basic effect $\mathbb{c}$. For our running example with only the single effect $\mathsf{partial}$, the $unit$ operation is defined as follows:

$$
\begin{aligned}
&unit : \tau \to \text{Partial}(\tau) \\
&unit(x) = \text{success}(x)
\end{aligned}
$$

Essentially $unit$ turns an effectless value into a successful value with the $\mathsf{partial}$ effect.

Now consider the semantic effect system in Figure 3. We define elements of $T_{\mathsf{nd}(n)}(\tau)$ as nonempty finite sets of up to n elements of

τ, indicating that a procedure with effect $\mathsf{nd}(n)$ does in fact produce at most n different values (and always produces at least one value). The primitive effectful operators $arb_n : \tau^n \xrightarrow{\mathsf{nd}(n)} \tau$ can be defined in terms of these functors via the following:

$$arb_n(x_1, \ldots, x_n) = \{x_1, \ldots, x_n\}$$

The operations $map_{\mathsf{nd}(n)}(f)$ simply apply f to each of the elements of a set in $T_{\mathsf{nd}(n)}(\tau)$. The *unit* operation gives constants the $\mathsf{nd}(1)$ effect by mapping them to the computation nondeterministically producing only that constant.

So far we have only addressed the basic components of nominal effect systems. Next we show how to give a semantics to nominal sequential composition and then nominal subeffects.

Compressing Effects Going back to the partial effect, consider a slightly more complex example: $(64 \div x) \div y$. Once again we can use the functor representation of the partial effect in order to formalize this expression:

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

The type of this formalization, though, is Partial(Partial($\mathbb{Z}$)), since the computation we mapped, namely $\lambda d.d \div y$, also has the partial effect. Although having a doubly partial value allows us to determine which $\div$ failed, typically we are only concerned with whether *any* $\div$ failed. Thus we want a way to *compress* the doubly partial value into a singly partial value. In this example we have a single sequentially compressive effect, so we can apply the category-theoretic concept of monadic joins:

$$join : \text{Partial}(\text{Partial}(\tau)) \to \text{Partial}(\tau)$$
$$join(ppx) = \mathbf{case}\ ppx \begin{cases} \text{success}(\text{success}(x)) & \mapsto \text{success}(x) \\ \text{success}(\text{failure}) & \mapsto \text{failure} \\ \text{failure} & \mapsto \text{failure} \end{cases}$$

Essentially *join* compresses a doubly partial effect by failing if either operation fails and otherwise forwarding the successful result. There is some loss of information, but typically this is information that would be more cumbersome to propagate and reason about than it is worth.

In a more complex nominal effect system, we need to give a semantics to sequences of procedures with *different* effects. Using the above techniques, if the first effectful procedure has effect ε and the second has effect ε', then mapping the second procedure to handle the first's effect results in an expression with type $T_\varepsilon(T_{\varepsilon'}(\tau))$. So we need a way to compress this doubly effectful value into a value with a single effect $\varepsilon \fatsemi \varepsilon'$. For this, we use a *sequential* semantic effect system.

Definition. A sequential semantic effect system for a sequential nominal effect system $\langle \text{EFF}, \mathbb{c}, \fatsemi \rangle$ additionally specifies for each pair of effects ε and ε' a join operation (specifically a natural transformation) describing how to compress these effects when used sequentially:

$$join_{\varepsilon,\varepsilon'} : T_\varepsilon(T_{\varepsilon'}(\tau)) \to T_{\varepsilon \fatsemi \varepsilon'}(\tau)$$

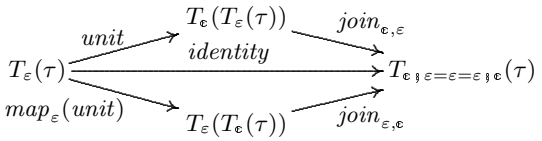
This family of join operations must satisfy the following two equational requirements in order for the semantics to be *coherent*, meaning all possible ways of inserting implicit semantic operations lead to equivalent semantics.

The first requirement is an associativity law. Suppose we have a triply effectful value of type $T_\varepsilon(T_{\varepsilon'}(T_{\varepsilon''}(\tau)))$. Again, we want to compress the effects into a singly effectful value of type $T_{\varepsilon \fatsemi \varepsilon' \fatsemi \varepsilon''}(\tau)$. However, we have two ways to do this:

$$\begin{array}{ccccc} & \xrightarrow{join_{\varepsilon,\varepsilon'}} & T_{\varepsilon \fatsemi \varepsilon'}(T_{\varepsilon''}(\tau)) & \xrightarrow{join_{\varepsilon \fatsemi \varepsilon', \varepsilon''}} & \\ T_\varepsilon(T_{\varepsilon'}(T_{\varepsilon''}(\tau))) & & & & T_{(\varepsilon \fatsemi \varepsilon') \fatsemi \varepsilon'' = \varepsilon \fatsemi (\varepsilon' \fatsemi \varepsilon'')}(\tau) \\ & \xrightarrow[map_\varepsilon(join_{\varepsilon',\varepsilon''})]{} & T_\varepsilon(T_{\varepsilon' \fatsemi \varepsilon''}(\tau)) & \xrightarrow[join_{\varepsilon,\varepsilon' \fatsemi \varepsilon''}]{} & \end{array}$$

The top path corresponds to compressing the later effects and then the earlier effects, while the bottom path corresponds to the opposite. We require these two methods to be equivalent. This way neither the programmer nor the compiler needs to worry about the order in which effect compression is done. Furthermore, this makes composing effectful procedures an associative process, and inlining effectful function calls a semantics-preserving transformation.

The other requirement is an identity law, requiring all paths in the following diagram to be equivalent to the identity function:



The top path corresponds to inserting the basic effect before another effectful procedure and then compressing the result. The bottom path corresponds to inserting the basic effect after another effectful procedure then compressing the result. Requiring both paths to be equivalent to the identity function indicates that we can insert the basic effect $\mathbb{c}$ anywhere using *unit* without impacting the semantics. This formalizes the intuition that $\mathbb{c}$ is the basic effect and *unit* gives effectless values a trivial form of the basic effect. ◆

Now once again consider the semantic effect system in Figure 3 for bounded non-determinism. The nominal sequential composition for this effect system says that if one line non-deterministically produces up to m values, and the next line up to n, then the combination produces up to $m * n$ values. The result of sequencing two lines without compression in this semantic effect system produces a value with semantic type $T_{\mathsf{nd}(m)}(T_{\mathsf{nd}(n)}(\tau))$, i.e. up to m sets each containing up to n elements of τ. The *join* operation simply takes the union of the m sets, resulting in a set with up to $m * n$ elements of τ (i.e. an element of $T_{\mathsf{nd}(m*n)}(\tau)$). Note that it is generally impossible to define a natural function of the form $T_{\mathsf{nd}(n)}(T_{\mathsf{nd}(n)}(\tau)) \to T_{\mathsf{nd}(n)}(\tau)$ should we prefer a more traditional sequentially compressive nominal effect system. Thus, the fact that bounded non-determinism can be captured by our framework relies on our framework's flexibility of *not* requiring all effects to be sequentially compressive.

Using the structure we have so far we can give a semantics to simple imperative programs such as the following generic program:

$x \leftarrow e;$	(has effect ε)
$y \leftarrow f(x);$	(has no effect)
`return` $g(x, y)$	(has effect ε')

We can use *unit* to incorporate the second line into our effectful system, *map* to propagate the effects, and *join* to compress the effects. There are multiple ways to translate this program depending on how we choose to compress the effects (subscripts implicit):

$$join(join(map(\lambda x.map(\lambda y.g(x,y))(unit(f(x))))(e)))$$
or
$$join(map(\lambda x.join(map(\lambda y.g(x,y))(unit(f(x)))))(e))$$

If x is not actually used by g, we could additionally translate to:

$$join(map(\lambda y.g(y))(join(map(\lambda x.unit(f(x)))(e))))$$

Nonetheless, the associativity law ensures that all of these translations are equivalent. Furthermore, the identity law enables us to simplify the translations by removing a use of *join* and *unit*:

$$join(map(\lambda x.g(x, f(x)))(e))$$

Thus even with multiple effects we ensure a coherent semantics.

Converting Effects Lastly we formalize the semantics of subeffects, which is done in a manner very similar to formalizing the semantics of subtypes.

Definition. A semantic subeffect system for a nominal subeffect system $\langle \text{EFF}, \mathbb{c}, \leq \rangle$ additionally specifies for each subeffect pair $\varepsilon \leq \varepsilon'$ a convert operation describing how to convert procedures with effect ε to procedures with effect ε':

$$convert_{\varepsilon \leq \varepsilon'} : T_\varepsilon(\tau) \to T_{\varepsilon'}(\tau)$$

This family of convert operations must satisfy the following two equational requirements (plus a third if the effect system also has sequential composition) in order for the semantics to be coherent.

First we require that converting from an effect to itself be trivial:

$$convert_{\varepsilon \leq \varepsilon} = identity$$

Second we require that all methods for converting one effect to another be equivalent. In particular, given a chain of subeffects $\varepsilon \leq \varepsilon' \leq \varepsilon''$, we also know $\varepsilon \leq \varepsilon''$ holds since $\leq$ is transitive; thus the following two paths must be equivalent:

$$T_\varepsilon(\tau) \xrightarrow{convert_{\varepsilon \leq \varepsilon'}} T_{\varepsilon'}(\tau) \xrightarrow{convert_{\varepsilon' \leq \varepsilon''}} T_{\varepsilon''}(\tau) \qquad T_\varepsilon(\tau) \xrightarrow{convert_{\varepsilon \leq \varepsilon''}} T_{\varepsilon''}(\tau)$$

Third, if the effect system also has sequential composition, then we require that the convert operations be compatible with the join operations. In particular, the following two programs must be equivalent for the semantics to be coherent:

$$\begin{array}{l} x \leftarrow convert_{\varepsilon_1 \leq \varepsilon_1'}(e_1); \\ \texttt{return}\ \ convert_{\varepsilon_2 \leq \varepsilon_2'}(e_2) \end{array} = convert_{\varepsilon_1 \fatsemi \varepsilon_2 \leq \varepsilon_1' \fatsemi \varepsilon_2'} \begin{pmatrix} x \leftarrow e_1; \\ \texttt{return}\ \ e_2 \end{pmatrix}$$

That is, the semantics must be the same whether implicit coercions occur before or after sequencing. This is accomplished by requiring the following two paths to be equivalent:

$$\begin{array}{ccccc} & & map_{\varepsilon_1'}(convert_{\varepsilon_2 \leq \varepsilon_2'}) & & \\ & convert_{\varepsilon_1 \leq \varepsilon_1'} \nearrow & T_{\varepsilon_1'}(T_{\varepsilon_2}(\tau)) \to T_{\varepsilon_1'}(T_{\varepsilon_2'}(\tau)) & \searrow\ join_{\varepsilon_1', \varepsilon_2'} & \\ T_{\varepsilon_1}(T_{\varepsilon_2}(\tau)) & & & & T_{\varepsilon_1' \fatsemi \varepsilon_2'}(\tau) \\ & join_{\varepsilon_1, \varepsilon_2} \searrow & T_{\varepsilon_1 \fatsemi \varepsilon_2}(\tau) & \nearrow\ convert_{\varepsilon_1 \fatsemi \varepsilon_2 \leq \varepsilon_1' \fatsemi \varepsilon_2'} & \end{array}$$

Thus joining after converting (the top path) must be equivalent to joining before converting (the bottom path). This requirement is both necessary and sufficient for subeffecting to be natural (i.e. satisfying the requirements put forth by Reynolds [20]). ◆

The convert operations for the bounded non-determinism semantic effect system in Figure 3 are trivial since every set containing up to m elements is already a set containing up to n elements when m is less than n. So at this point we present in Figure 4 the semantic effect system for the nominal effect system for state machines that we presented in Figure 1. This semantic effect system is much more intricate than that for bounded non-determinism because the effects vary much more in their meaning. In particular the convert functions, although simple, are not trivial, and they provide some sense of how the pieces of this effect system fit together. One can also see that a $\mathsf{read}(S)$ after a $\mathsf{set}(S)$ can actually be combined into just the $\mathsf{set}(S)$ effect. Next we examine various ways to classify properties of semantic effect systems in much the same way we did with nominal effect systems.

3.2 Classifying Semantic Effect Systems

As part of our goal towards improving the understanding of effects, here we identify some abstract properties of semantic effect systems that we find interesting. In Section 5 we will identify valuable properties of individual effects, but here we focus on properties which provide a means for evaluating nominal effect systems.

Efficiency We define a notion of *efficiency*, which as we will see, provides a way to evaluate the accuracy of effect systems. We say an operation in a nominal effect system is *efficient* if it has a semantic effect system such that the corresponding semantic operation is surjective in some sense. For example, the primitive operations $arb_n : \tau^n \xrightarrow{\mathsf{nd}(n)} \tau$ are efficient because each set in $T_{\mathsf{nd}(n)}(\tau)$ can directly result from a use of arb_n. This would not be the case if arb_n had effect $\mathsf{nd}(n+1)$ instead, since sets with $n+1$ elements would never occur. In particular, the fact that Haskell gives the same $\mathsf{state}(S)$ effect to all the stateful primitive operations makes these operators inefficient, formalizing the imprecision in their single-effect monad abstraction.

ε	$T_\varepsilon(\tau)$	$map_\varepsilon(f)$
$\mathbb{c}$	τ	f
$\mathsf{read}(S)$	$S \to \tau$	$\lambda r.\lambda s.f(r(s))$
$\mathsf{set}(S)$	$\tau \times S$	$\lambda\langle x, s\rangle.\langle f(x), s\rangle$
$\mathsf{update}(S)$	$\tau \times (S \to S)$	$\lambda\langle x, u\rangle.\langle f(x), u\rangle$
$\mathsf{state}(S)$	$S \to (\tau \times S)$	$\lambda m.\lambda s.\mathbf{let}\ \langle x, s'\rangle = m(s)\ \mathbf{in}\ \langle f(x), s'\rangle$

$$unit = identity$$

Convert Operations: $\varepsilon \xrightarrow{convert_{\varepsilon \leq \varepsilon'}} \varepsilon'$

$$\begin{array}{ccccc} & & \lambda x.\lambda s.x & & \\ & \mathbb{c} & \longrightarrow & \mathsf{read}(S) & \\ \lambda x.\langle x, \lambda s.s\rangle & \searrow & & \searrow & \lambda r.\lambda s.\langle r(s), s\rangle \\ \mathsf{set}(S) & \longrightarrow & \mathsf{update}(S) & \longrightarrow & \mathsf{state}(S) \\ & \lambda\langle x, s'\rangle.\langle x, \lambda s.s'\rangle & & \lambda\langle x, u\rangle.\lambda s.\langle x, u(s)\rangle & \end{array}$$

Incomplete Table of Join Operations

ε	ε'	$T_\varepsilon(T_{\varepsilon'}(\tau))$	$join_{\varepsilon, \varepsilon'}$
$\mathsf{read}(S)$	$\mathsf{read}(S)$	$S \to (S \to \tau)$	$\lambda r.\lambda s.r(s)(s)$
$\mathsf{read}(S)$	$\mathsf{set}(S)$	$S \to (\tau \times S)$	$identity$
$\mathsf{set}(S)$	$\mathsf{read}(S)$	$(S \to \tau) \times S$	$\lambda\langle r, s\rangle.\langle r(s), s\rangle$
$\mathsf{set}(S)$	$\mathsf{set}(S)$	$(\tau \times S) \times S$	$\lambda\langle\langle x, s'\rangle, s\rangle.\langle x, s'\rangle$
$\mathsf{set}(S)$	$\mathsf{update}(S)$	$(\tau \times (S \to S)) \times S$	$\lambda\langle\langle x, u\rangle, s\rangle.\langle x, u(s)\rangle$
$\mathsf{set}(S)$	$\mathsf{state}(S)$	$(S \to (\tau \times S)) \times S$	$\lambda\langle m, s\rangle.m(s)$
$\mathsf{update}(S)$	$\mathsf{set}(S)$	$(\tau \times S) \times (S \to S)$	$\lambda\langle\langle x, s\rangle, u\rangle.\langle x, s\rangle$

Figure 4. Semantic Effect System for State Machines

A subeffect system has efficient joins if, whenever ε and ε' have a join $\varepsilon \sqcup \varepsilon'$ in the subeffect preorder, the pair of operations $\langle convert_{\varepsilon \leq \varepsilon \sqcup \varepsilon'}, convert_{\varepsilon' \leq \varepsilon \sqcup \varepsilon'} \rangle$ forms what is known as an *extremal epi-sink* [1]; that is, every element of $T_{\varepsilon \sqcup \varepsilon'}(\tau)$ is in the image of at least one of the two convert operations. If a nominal subeffect system has efficient joins, then that means there is essentially no significant loss of information at merge points such as if-then-else statements or pattern-match expressions since, loosely speaking, each effectful value with the merged effect can occur from one of the branches/cases. The bounded non-determinism effect system has this property, but the state machine effect system does not: the join of $\mathsf{read}(S)$ and $\mathsf{set}(S)$ is $\mathsf{state}(S)$, yet there are many procedures with the $\mathsf{state}(S)$ effect which do more than just read the state or just set the state.

Sequential composition is efficient if each *join* operator is surjective (specifically an *extremal epimorphism* [1]), signifying that there is no significant loss of information due to sequencing effects. Both the bounded non-determinism and state machine effect systems have this property. Interestingly, this would not be the case had we defined $\mathsf{set}(S) \fatsemi \mathsf{read}(S)$ as $\mathsf{state}(S)$ instead in order to make sequential composition increasing. Thus using this notion of efficiency in semantics provides a means to evaluate the accuracy of the nominal effect system.

Losslessness We say an operation in a semantic effect system is lossless if it is undoable (specifically a *section* [1]). For example, the compressing *join* for the $\mathsf{read}(S)$ effect followed by the $\mathsf{set}(S)$

effect resulting in the $\texttt{state}(S)$ effect is lossless (better yet it is an isomorphism). More significantly, a semantic subeffect system is lossless if each *convert* operation is lossless. Both the bounded non-determinism and state machine effect systems have this property; however it need not always hold. Consider an effect system with two kinds of exception effects: deterministic exceptions $\texttt{exc}(E)$ and non-deterministic exceptions $\texttt{nd-exc}(E)$ (where E is the set of possible exceptions). It is quite reasonable to have $\texttt{exc}(E)$ be a subeffect of $\texttt{nd-exc}(E)$. However, this conversion is lossy, as an arbitrary non-deterministic process cannot be made deterministic in a way that undoes the conversion. Nonetheless, since many semantic subeffect systems are lossless, it is a good property to be aware of when reasoning about effects.

3.3 Category-Theoretic Formalization

Above we have introduced a number of concepts, operations, and requirements. In doing so we have significantly generalized traditional monads, which essentially formalize a solitary effect, to systems capable of formalizing multiple interacting effects. Here we use category theory to demonstrate that we accomplished precisely this without imposing any unnecessary constraints while imposing precisely the constraints required to get a coherent semantics. We have determined that the entire definition of a framework can be described in a single sentence using simple concepts from 2-category theory. This suggests that the specification in our framework is in fact very natural, even if it seems complex. Furthermore, from this definition we can prove that our specification is exactly precise and general enough to formalize effectful computations with sequential composition. Also, the definition informs us how to formally compare effect systems. Below we give the formal definitions and theorems using 2-category theory for sake of completeness. Unfortunately we do not have the space to explain 2-category theory, but the material below is not necessary for understanding the rest of the paper, should the reader opt to bypass it.

Definition. A sequential effect system with subeffects is a lax functor from a one-object locally thin 2-category (i.e. a monoid with a congruent preorder) to the 2-category of categories $\mathbf{CAT}$.◆

This succinctly formalizes all the operations and equality requirements of our framework and may be familiar since a monad is a lax functor from $\mathbf{1}$ to $\mathbf{CAT}$. EFF is the set of morphisms in the domain. $\mathbb{c}$ is the identity morphism. $\fatsemi$ is (the opposite of) composition of these morphisms. $\varepsilon \leq \varepsilon'$ holds when there is a (unique) 2-cell from morphism ε to morphism ε'. The functors, unit operation, join operations, and convert operations are specified by the lax functor. The equality requirements are the coherence requirements.

Now we define the category of computations in a given semantic effect system. Suppose we have a sequential effect system $E = \langle \text{EFF}, \mathbb{c}, \fatsemi, \leq, T, \mathit{map}, \mathit{unit}, \mathit{join}, \mathit{convert} \rangle$ on category $\mathbf{Type}$. We formalize the 2-category of effectful computations $\mathcal{K}_E$ as follows:

- Objects are the objects τ of $\mathbf{Type}$.
- Morphisms from τ to τ' are morphisms τ to $T_\varepsilon(\tau')$ in $\mathbf{Type}$ for *some* effect ε in EFF.
- The identity of each object τ is $\mathit{unit} : \tau \to T_{\mathbb{c}}(\tau)$.
- Composition of morphisms uses the join operations as before.
- There is a 2-cell from $f : \tau \to T_\varepsilon(\tau')$ to $g : \tau \to T_{\varepsilon'}(\tau')$ when ε is a subeffect of ε' and g equals $\mathit{convert}_{\varepsilon \leq \varepsilon'} \fatsemi f$.

This forms a 2-category precisely because of our equality requirements. Those familiar with monads may know of the Kleisli category $\mathcal{K}_M$ for a monad M [12] and its connection to effectful computations [18]. In fact, each monad is a lax functor from $\mathbf{1}$ to $\mathbf{CAT}$, and the construction above applied to this lax functor coincides with the Kleisli category. This demonstrates that we are consistent with earlier formalizations of effectful computations.

Next we want to show that our definition of sequential effect systems is exactly precise and general enough to represent effectful computation with sequential composition. This is formalized by the following theorem stating that the category of computations satisfies a couniversal property with respect to the effect system. This theorem is straightforward to prove so we do not do so here.

Theorem. *$\mathcal{K}_E$ is the lax colimit [6, 21] of the lax functor E.*

In particular, the Kleisli category of a monad is the lax colimit of its corresponding lax functor. Thus the above theorem simply indicates that our generalization of monads preserved the important properties for representing effectful computations.

Lastly, we formalize how to compare sequential effect systems.

Definition. A morphism from a sequential effect system E to a sequential effect system E' is a functor $F : \mathit{dom}(E) \to \mathit{dom}(E')$ paired with a colax natural transformation $\phi : E \Rightarrow E' \circ F$. ◆

Such a morphism induces a functor from $\mathcal{K}_E$ to $\mathcal{K}_{E'}$, justifying our definition. We could go into more detail, but at this point we continue on so that we may compare our framework with traditional monadic techniques. We have already formalized using 2-category theory that a monad defines the semantics of a nominal effect system with only one effect, but next we use simpler formalisms to classify the nominal effect systems whose semantics can be formalized by a *hierarchy* of monads.

4. Monadic Semantics

Monads have been the traditional approach for formalizing the semantics of effects. In our framework, a monad arises from the special case when the system has only one effect (using the category-theoretic perspective of map, join, and unit rather than the functional-languages perspective of `bind` and `return`). Thus our framework generalizes monads from a single effect to multiple effects. However, there have been other approaches to formalizing multiple effects by using multiple monads. Again, these arise as special cases in our framework, but we have also seen effect systems which do not fall within these special cases. Here we classify which nominal effect systems have their semantics formalized using a hierarchy of monads, then illustrate why the counterexamples do not fall within this classification and how this can actual be a benefit. In the section afterwards, we will present extensions to our framework for handling additional roles effects play in a language are not addressed by traditional monadic techniques.

4.1 The Marriage of *Analytical* Effects and Monads

Not only does the semantic effect system reflect upon the nominal effect system, but the nominal effect system significantly impacts the semantic effect system. Wadler and Thiemann proposed, but did not prove, a marriage between effect systems and hierarchies of monads and monad morphisms [27], demonstrating that the semantics of a *specific* type-and-effect system [22] can be formalized using a hierarchy of monads, and claiming "it seems clear that any effect system can be adapted to monads in a similar way". However, as we have shown, not all nominal effects have a natural and efficient semantic definition that can be represented by a monad. In particular, the bounded non-determinism effects have no monadic join, and the $\texttt{set}(S)$ effect has no monadic unit. However, there are still many effect systems for which Wadler and Thiemann's claim holds. Using our framework we can classify precisely which nominal effect systems are represented by a hierarchy of monads and monad morphisms. First we classify precisely which individual nominal effects are represented by a monad.

Lemma. *The denotational semantics of a nominal effect is formalized by a monad if and only if it is sequentially compressive and can soundly be made insertable.*

Proof. We prove here only the if since the other direction is less enlightening. A monad M is a tuple $\langle T_M, map_M, unit_M, join_M \rangle$ where $\langle T_M, map_M \rangle$ is a functor, $unit_M$ is an operation $\tau \to T_M(\tau)$, and $join_M$ is an operation $T_M(T_M(\tau)) \to T_M(\tau)$ satisfying certain equality requirements. The monad corresponding to an insertable sequentially compressive effect ε uses $\langle T_\varepsilon, map_\varepsilon \rangle$ as its functor. The $unit_M$ operation is defined as $convert_{\mathbb{c} \leq \varepsilon} \circ unit$. The $join_M$ operation is defined as $join_{\varepsilon,\varepsilon}$ (which is correctly typed since ε is sequentially compressive meaning $\varepsilon \fatsemi \varepsilon = \varepsilon$). These operations satisfy the equality requirements for monads due to the equality requirements of semantic effect systems. □

Additionally, one can easily prove that a convert operation between two insertable sequentially compressive effects satisfies the equality requirements of semantic effect systems if and only if it forms what is known as a colax map of monads in the category theory community [13] or a monad morphism in the functional languages community [2, 18, 25]. This fact, combined with the above lemma, entails the following corollary.

Corollary. *The denotational semantics of a sequential nominal effect system with subeffects is formalized by a hierarchy of monads and monad morphisms if and only if it is sequentially compressive and $\mathbb{c}$ can soundly be made a subeffect of all effects.*

Analytical effect systems typically use a join semi-lattice of effects. As such, using the join operator $\sqcup$ as sequential composition $\fatsemi$ is essentially trademark for such systems. Recognizing this, using the above corollary we can now prove that Wadler and Thiemann's claimed marriage between effects and monads holds particularly for *analytical* effect systems.

Theorem. *The denotational semantics of any analytical effect system is formalized by a hierarchy of monads and monad morphisms because for sequential composition $\fatsemi$ they use the join operator $\sqcup$ on their join semi-lattice of nominal effects.*

Proof. The join operator $\sqcup$ is idempotent by nature, and so the nominal effect system is sequentially compressive. $\mathbb{c}$ is always the bottom of this lattice since $\mathbb{c}$ is always a subeffect of $\mathbb{c} \sqcup \varepsilon$ by definition of join, and $\mathbb{c} \sqcup \varepsilon$ equals ε since it equals $\mathbb{c} \fatsemi \varepsilon$ by assumption and this equals ε because $\mathbb{c}$ is the identity element for $\fatsemi$ by requirement. Thus these nominal effect systems satisfy the criteria of the above corollary. □

4.2 Non-Monadic Semantic Effect Systems

Using our framework we have demonstrated the significant impact the structure of a nominal effect system has on the structure of its semantics. Because they use a join semi-lattice for *nominal* sequential composition, the *semantics* of analytical type systems are always formalized using monads. However, the semantic effect systems in Figures 3 and 4 are not monads. This is because the bounded non-determinism nominal effects are not sequentially compressive, and the $\mathtt{set}(S)$ nominal effect is not insertable.

Interestingly, not only does the flexibility of our framework allow us to express more effects than monads, it also allows us to formalize more complex interactions between effects. In particular, in analytical effect systems nominal sequential composition is always increasing and sequentially compressive due to the nature of joins on preorders. Because of these two properties, the $join$ operations for different effects are always determined completely by the $join$ operations for effects with themselves and the $convert$ operations between effects, as demonstrated by the following theorem.

Theorem. *Semantic sequential composition for different effects in increasing sequentially compressive nominal effect systems is determined by the convert operations and semantic sequential composition of effects with themselves:*

$$join_{\varepsilon,\varepsilon'} = join_{\varepsilon \fatsemi \varepsilon', \varepsilon \fatsemi \varepsilon'} \circ convert_{\varepsilon \leq \varepsilon \fatsemi \varepsilon'} \circ map_\varepsilon(convert_{\varepsilon' \leq \varepsilon \fatsemi \varepsilon'})$$

Proof. The left side is equal to $convert_{\varepsilon \fatsemi \varepsilon' \leq \varepsilon \fatsemi \varepsilon'} \circ join_{\varepsilon,\varepsilon'}$ since $convert_{\varepsilon \fatsemi \varepsilon' \leq \varepsilon \fatsemi \varepsilon'}$ converts an effect to itself and so must be the identity function for semantics to be coherent. That in turn is equivalent to the right side because the join and convert operations must be compatible for semantics to be coherent. □

In other words, in such effect systems the interactions between different effects is only containment of effects. In fact, monadic techniques for building effect systems handle interaction of two different effects by combining their monads into one monad containing both [7, 11, 14, 16]. On the other hand, the bounded non-determinism system is not sequentially compressive, and the state machine system is not increasing, so they can define more interesting interactions, such as that for $\mathtt{read}(S)$ after $\mathtt{set}(S)$. Thus, not only is our framework able to formalize more complex effect systems, but it can also formalize more complex interactions of effects.

5. Extending Effect Systems

So far we have focused on the role of effects in simple imperative languages using subeffects and sequential effects since these are the roles traditional monadic techniques have addressed. In this section, we identify other important roles effects play in realistic languages and extend our semantic framework to account for these additional roles. First, we add a lateral form of composition, as opposed to sequential composition, which is particularly important for languages in which subexpressions can be effectful. Second, we recognize that certain effects have some degree of flexibility in their semantics which is important to properly understanding these effects, with non-determinism being the primary example. Third, we identify the additional structure necessary for effects to be used in infinite processes such as unbounded loops and recursion. These additions make our framework capable of formalizing the semantics of effects in a large class of imperative and functional languages. In the section afterwards, we will apply these extensions to many stages of the language design and implementation process.

5.1 Lateral Composition

Consider the two effectful integer expressions $(128 \div x) \div y$ and $(128 \div x) + (128 \div y)$ which both have two effectful operations. In the first expression, the effectful operations occur in sequence and we have already shown how to give this a semantics. In the second expression however, the effectful operations occur side-by-side; neither $\div$ depends on the result of the other and so there is no need to restrict ourselves to applying them sequentially. To address this, we introduce what we call *lateral* composition.

Definition. Nominal lateral composition is an associative binary operator $\fatcomma : \text{EFF} \times \text{EFF} \to \text{EFF}$ with $\mathbb{c}$ as the identity element. For typed nominal effect systems, $\fatcomma$ can be a partial operator. Should a nominal effect system have both subeffects and lateral composition, then lateral composition must preserve subeffects: $\varepsilon_1 \leq \varepsilon_1' \wedge \varepsilon_2 \leq \varepsilon_2' \implies \varepsilon_1 \fatcomma \varepsilon_2 \leq \varepsilon_1' \fatcomma \varepsilon_2'$.

Semantic lateral composition specifies merge operations describing how to combine ordered pairs of effectful values into an effectful pair of values:

$$merge_{\varepsilon,\varepsilon'} : T_\varepsilon(\tau) \times T_{\varepsilon'}(\tau') \to T_{\varepsilon \fatcomma \varepsilon'}(\tau \times \tau')$$

This family of merge operations must satisfy various equational requirements, which we do not formally specify here. In short, in

the same way semantic sequential composition generalizes monads from one effect to multiple effects, semantic lateral composition generalizes *lax monoidal functors* [13] from one effect to multiple effects. As before, these requirements are necessary and sufficient for the semantics to be coherent. ◆

There are some very common forms of lateral composition for a sequential effect system. For any two effectful expressions, we can do a left-to-right evaluation:

$$ltor_{\varepsilon,\varepsilon'}(e,e') = join_{\varepsilon,\varepsilon'}(map_{\varepsilon}(\lambda x.map_{\varepsilon'}(\lambda y.\langle x,y\rangle)(e'))(e))$$

A language has left-to-right evaluation of arguments when $\fatsemi$ equals $\fatsemi$ and $merge$ equals $ltor$. Similarly a language has right-to-left evaluation of arguments when $\varepsilon \fatsemi \varepsilon'$ equals $\varepsilon' \fatsemi \varepsilon$ (note the reverse order) and $merge$ equals $rtol$ (the reverse of $ltor$).

Not all languages will have $\fatsemi$ coincide with $\fatsemi$ (or its reverse), although often the two will be related. Consider our earlier example of deterministic exceptions $\mathsf{exc}(E)$ and non-deterministic exceptions $\mathsf{nd\text{-}exc}(E)$. A language may decide to have $\mathsf{exc}(E) \fatsemi \mathsf{exc}(E)$ be $\mathsf{exc}(E)$, since when sequencing exception-throwing procedures the exceptional behavior is clear. However, it may decide to have $\mathsf{exc}(E) \fatsemi \mathsf{exc}(E)$ be $\mathsf{nd\text{-}exc}(E)$ because, if two side-by-side processes both throw an exception, it is not clear which of the two exceptions should be propagated. Thus this language purposely has $\fatsemi$ and $\fatsemi$ differ so that the programmer can express when they actually care that exceptions are deterministic. Note that in this example $\varepsilon \fatsemi \varepsilon'$ is always a subeffect of $\varepsilon \fatsemi \varepsilon'$.

Lateral composition can also be useful for effects which may not have an intuitive sequential composition. For example, we can consider n-length vectors as a *laterally* compressive effect ($\varepsilon \fatsemi \varepsilon = \varepsilon$). This has an obvious merge operation: given a pair of vectors simply construct a vector of pairs componentwise. Also, given an effectless value and a vector, simply construct a vector of pairs whose first component is always that effectless value. This lateral effect system produces the vector semantics we are all familiar with for expressions such as $5 * \vec{x} + 10 * \vec{y} - 1$. Furthermore this lateral effect system is *symmetric*: $\varepsilon \fatsemi \varepsilon'$ always equals $\varepsilon' \fatsemi \varepsilon$ and

$$merge_{\varepsilon',\varepsilon}(e',e) = map_{\varepsilon \fatsemi \varepsilon'}(\lambda\langle x,y\rangle.\langle y,x\rangle)(merge_{\varepsilon,\varepsilon'}(e,e'))$$

That is, the order of arguments does not matter effectwise. A similar symmetric lateral effect system can also be made for databases, and it is this structure that makes these domains so amenable to data parallelism. Later we will show how semantic lateral composition and *flexible* effects can be applied to thread parallelism as well.

5.2 Flexible Effects

Consider once again the effect system with deterministic and non-deterministic exceptions. We specified $\mathsf{exc}(E) \fatsemi \mathsf{exc}(E)$ as $\mathsf{nd\text{-}exc}(E)$ so that the user could inform the compiler when any order of evaluation is valid. However, transforming the program so that these ambiguous cases evaluate left-to-right is not strictly semantics-preserving since doing so actually makes the program *more* deterministic. In settings which use *enumerable* non-determinism, such as in parsers, it is important to strictly preserve non-determinism. However in our intended setting of non-deterministic exceptions, we introduced this non-determinism to give the compiler the *flexibility* to choose in which order to evaluate the arguments; we call this *flexible* non-determinism.

Thus, we introduce a notion of *flexible* semantic effects in order to formalize the difference between effects such as flexible non-determinism and enumerable non-determinism, and to formalize when transformations need not *strictly* preserve semantics. A flexible effect ε specifies a preorder $\preccurlyeq_\varepsilon$ on $T_\varepsilon(\tau)$. This preorder must be compatible with the various effect structures, but we do not digress into the specific requirements. In short, simply add another dimension to the categorical formalization. $e \preccurlyeq_\varepsilon e'$ essentially indicates that e is less flexible (e.g. more deterministic) than e'. In this situation, the compiler is allowed to replace e' with e (or treat e as e'); thus $\preccurlyeq$ is a lot like subtypes but for expressions with the same effect. The difference between flexible non-determinism and enumerable non-determinism, then, is that $\preccurlyeq$ for flexible non-determinism is non-trivial (corresponding to the subset relation) while $\preccurlyeq$ for enumerable non-determinism is simply equality of subsets.

Let us return to our effect system for exceptions. When we specified the lateral composition structure, we did so with flexible non-determinism in mind. So we say that $e \preccurlyeq_{\mathsf{nd\text{-}exc}(E)} e'$ holds when the exceptional behavior of e is more deterministic than that of e' and otherwise they are identical. Using this notion of flexibility we can then observe that $convert \circ ltor \preccurlyeq merge$ and $convert \circ rtol \preccurlyeq merge$ always hold. This shows why in this effect system the compiler may choose any argument-evaluation order.

5.3 Infinite Effects

Many interesting programs have the potential to run forever, essentially taking an infinite number of effectful steps. In fact, the effect itself may determine whether an execution runs forever, since a loop guard may depend on the user's input or even on a random coin flip. However, monadic structure is only capable of formalizing effects in *finite* processes for the same reason that monoids can only aggregate finite lists. Here we identify the additional structure necessary to formalize the behavior of effects in infinite processes such as unbounded loops or recursion.

To understand the challenge, consider the following program:

```
for (i = 0; check(i); i++)
  println(i);
```

Suppose we model the output effect as the list of strings to print. Then standard monadic translation would result in the following:

```
for (i = 0; check(i); i++)
  output = cons(toString(i), output);
return output;
```

This works provided the program terminates. However, should the program diverge then the output is never returned and in a sense never made visible to the user, which is not the semantics we would expect. Here we show how to give an effect system additional structure so that one can formalize the semantics intended for the above situation, in which output is made visible even if the program never terminates.

We begin with an introduction to infinite processes. Suppose we have a machine with an infinite number of states, such as a Turing machine (including the state of the tape). The mechanisms of a Turing machine define for each state whether it should advance to a new state (say by manipulating the tape) or whether it should halt and return an answer. Thus, an infinite process returning an answer in A has some set of states S and a function $advance : S \to S + A$ (where $+$ means or). It is fairly straightforward to see how an effectless while-loop could be expressed as such an infinite process.

We want to encapsulate the result of running such infinite processes as an effect. So, we need some functor T such that for any effectless infinite process $S \xrightarrow{advance} S + A$ we can map each state S to a result $T(A)$ indicating it returns something in A or runs forever. We can do this using a coinductive (i.e. infinite) data type:

$$\text{Coinductive Process}(A) = \text{process}(\text{Process}(A)) \mid \text{finish}(A)$$

If a state takes two steps then returns 5, that state would map to process(process(finish(5))). Any state which never leads to returning a value maps to the infinite structure process(process(...)).

This serves as a starting point, but we need to extend this to effectful infinite processes. The first observation we make is that, for any infinite process of the form $S \xrightarrow{advance} \text{Process}(S + A)$, there

is also an encapsulating map from S to Process(A). Thus a state machine which may take forever to decide whether it continues or finishes can also be represented by the Process functor. This corresponds to a while-loop whose guard may take forever to evaluate. With this observation we can generalize to effects more broadly.

Definition. A sequential nominal effect system with subeffects and infinite effects specifies a unary operator $\infty : \text{EFF} \to \text{EFF}$ indicating for each ε the insertable sequentially compressive effect $\infty(\varepsilon)$ to be used as the nominal effect of an infinite process with effect ε. For typed effect systems, ∞ can be a partial operator. We require $\varepsilon \fatsemi \infty(\varepsilon)$ to equal $\infty(\varepsilon)$. Also ∞ must preserve subeffects: $\varepsilon \leq \varepsilon' \implies \infty(\varepsilon) \leq \infty(\varepsilon')$. These are actually overly restrictive requirements, but they illustrate the key concepts.

A semantic effect system for such a nominal effect system specifies for each effect ε an operation $process_\varepsilon$ defining how to encapsulate infinite processes with effect ε as procedures with effect $\infty(\varepsilon)$:

$$process_\varepsilon : (\tau \to T_\varepsilon(\tau + \tau')) \to (\tau \to T_{\infty(\varepsilon)}(\tau'))$$

As usual this family of process operations has to satisfy various equational properties, but here we only emphasize one. For all effectful infinite processes $advance : \tau \to T_\varepsilon(\tau + \tau')$ the following two paths must be equivalent:

$$\begin{array}{ccc} \tau & \xrightarrow{process_\varepsilon(advance)} & T_{\infty(\varepsilon)}(\tau') \\ {\scriptstyle advance}\downarrow & & \uparrow{\scriptstyle join_{\varepsilon,\infty(\varepsilon)}} \\ T_\varepsilon(\tau + \tau') & \xrightarrow{\qquad\qquad} & T_\varepsilon(T_{\infty(\varepsilon)}(\tau')) \end{array}$$

$$map_\varepsilon \left(\lambda sa.\mathbf{case}\ sa \begin{cases} inl(s) \mapsto process_\varepsilon(advance)(s) \\ inr(a) \mapsto convert_{\mathbf{e} \leq \infty(\varepsilon)}(unit(a)) \end{cases} \right)$$

This property simply says that taking one step and then processing the whole is the same thing as processing the whole immediately. Thus loop peeling would be a semantics-preserving transformation as we would expect. Infinite effects can also be used to give a semantics to infinite recursion and this rule makes the standard fold and unfold transformations semantics-preserving. ◆

There are many infinite effects, but the simplest is the divergent effect defined by taking a quotient type of the Process data type:

$$T_{\mathsf{divergent}}(\tau) = {}^{\text{Process}(\tau)}\!/_{\text{process}(x)=x}$$

This means that any Process value is made equivalent to itself plus one more step, so that the quotient satisfies the above equality requirement. Furthermore, divergent is *infinitely compressive*: we can define $\infty(\mathsf{divergent})$ as divergent.

Some effects, such as the output effect, have no impact on the control flow of the program. For such effects we can define a function of the form $T_\varepsilon(\tau + \tau') \to T_\varepsilon(\tau) + T_\varepsilon(\tau')$ essentially pulling the branch out of the effect. For these effects (and only these effects), it is possible to define the semantics of $\infty(\varepsilon)$ as $T_{\mathsf{divergent}}(T_\varepsilon(\tau))$. In particular, this semantics applied to the output effect would indicate that outputs are only *externally* visible if the process terminates. However, we could also define the semantics of $\infty(\mathsf{output})$ by using coinductive lists of strings indicating that outputs are externally visible even if the process fails to terminate. There is a similar choice to be made for writes to memory, which is important for realistic systems which essentially use the memory abstraction for both internal storage and to communicate with the outside world. Interestingly both variants could be used as the denotational semantics for the semantics specified by Wadler and Thiemann in their example effect system [27] because they use a *small-step* operational semantics that is ambiguous as to what is and is not visible during a non-terminating process. This illustrates an interesting ambiguity in small-step operational semantics made clear by using denotational semantics.

6. Applications

So far we have shown various ways of understanding what effects are and their semantics. In this section we illustrate how this improved understanding can be applied to a variety of stages in the language design and implementation process besides semantics. We apply our framework to enable sound type generalization in the presence of effects, apply basic program optimizations in the presence of effects, and even run computations in parallel in the presence of effects. Furthermore, each application specifies its requirements on effects in a language-independent manner.

6.1 Type Generalization

Type generalization is not always sound in the presence of effects. Consider the following classic example:

$$\begin{array}{l} r \leftarrow \text{newRef}(\text{nil}); \\ \text{writeRef}(r, \text{cons}(\texttt{"blah"}, \text{nil})); \\ \texttt{return}\ \ \text{head}(\text{readRef}(r)) + 1 \end{array}$$

If we allowed standard type generalization, then we could give r the type $\forall\alpha.\text{Ref}(\text{List}(\alpha))$, which would allow the above code to type check. However, the above code is unsound using the standard semantics since it ends up treating a string as an integer.

Type generalization is not always unsound though. For example, the value restriction [28] allows type generalization when no effects are present. Furthermore, there are techniques for occasionally relaxing the value restriction even in the presence of effects [3]. These identify expressions that can be generalized regardless of the effect present, such as the following program expression:

$$\{x \leftarrow \text{newRef}(\texttt{"ignore"});\ \texttt{return}\ (\lambda x.x)\}$$

With our framework we can see that the semantic type of this expression is $T_\varepsilon(\forall\alpha.\alpha \to \alpha)$, where the polymorphism is already *inside* the effect, which is why type generalization is sound regardless of the effect.

Our framework can be used to identify *effects* for which type generalization is sound regardless of the *value*, an approach orthogonal to prior techniques. This question boils down to whether the effect ε has a function in the semantic domain mapping inhabitants of $\forall\alpha.T_\varepsilon(\tau)$, where the polymorphism is *outside* the effect, to inhabitants of $T_\varepsilon(\forall\alpha.\tau)$, where the polymorphism is *inside* the effect, satisfying additional equalities for coherence which we do not discuss here. Determining whether such a function exists can be challenging. However, in languages with common parametric polymorphism, there is a simple class of effects for which type generalization is always sound.

Suppose an expression has just the (deterministic) exception effect. Repeated evaluations of that expression will repeatedly produce the same value or exception. Informally, the following two programs are equivalent when e has the exception effect:

$$\{\texttt{return}\ \langle e, e\rangle\} = \{x \leftarrow e;\ \texttt{return}\ \langle x, x\rangle\}$$

This is not the case for effects such as $\mathsf{update}(S)$ and $\mathsf{state}(S)$, and particularly not for the memory allocation effect: evaluating newRef(nil) twice produces two distinct references.

This concept is similar to that of idempotent monads in the functional languages community [11], but the term idempotent is overloaded and particularly idempotent monad means something else in the category theory community [6], so instead we say the exception effect *preserves diagonals*. The diagonal function $\Delta : \tau \to \tau \times \tau$ simply duplicates a value: $\lambda x.\langle x, x\rangle$. A laterally compressive effect ε preserves diagonals if the following two paths are equivalent:

$$T_\varepsilon(\tau) \xrightarrow{\Delta} T_\varepsilon(\tau) \times T_\varepsilon(\tau) \xrightarrow{merge_{\varepsilon,\varepsilon}} T_\varepsilon(\tau \times \tau)$$
$$T_\varepsilon(\tau) \xrightarrow[map_\varepsilon(\Delta)]{} T_\varepsilon(\tau \times \tau)$$

That is, making two copies of an effectful expression and then evaluating both copies (the top path) produces the same result as evaluating once and duplicating the value (the bottom path).

In languages with common parametric polymorphism, type generalization is always sound for expressions with an effect which preserves diagonals. The intuition is that, if we evaluated the expression once for each type, the result would be the same for each type since the effect preserves diagonals. Thus the value resulting from a single evaluation does in fact belong to every type.

In fact, we can weaken our restriction by incorporating our concept of flexibility. A laterally compressive effect ε *laxly* preserves diagonals if $map_\varepsilon(\Delta) \preccurlyeq_\varepsilon merge_{\varepsilon,\varepsilon} \circ \Delta$, meaning it is valid to replace a program which evaluates the same expression twice with a program that only evaluates it once and then duplicates the value. For example, the flexible non-determinism effect laxly preserves diagonals because evaluating a non-deterministic expression once and then duplicating the value is more deterministic than evaluating the same non-deterministic expression twice. Now we can state our type generalization theorem using concepts from our framework.

Theorem. *In a language with common parametric polymorphism, type generalization is always sound for expressions with a laterally compressive effect that laxly preserves diagonals.*

Thus we can apply type generalization when the expression might throw exceptions, may not terminate, is flexibly non-deterministic, only reads from the heap, or even only writes to the heap. Even though it was relatively easy to show informally that the theorem holds, a full formal proof requires complex machinery such as fibred-category theory just to state what common parametric polymorphism is in a language-independent manner [5], so we do not present it here.

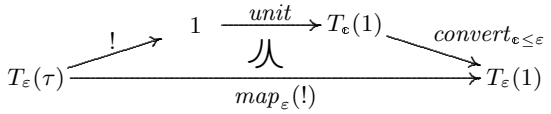
6.2 Program Optimizations

The compiler needs to understand a lot about the effects present in a program in order to optimize it. Many analyses can be viewed as determining which familiar effects are present in various subcomputations of the program so that the compiler can apply prior knowledge about these effects. For example, alias analyses are often used to determine whether a write-to-the-heap effect *commutes* with a read-from-the-heap effect, meaning they can be reordered.

At present, before applying an optimization a typical compiler determines which familiar effects are or are not present in order to ensure preservation of semantics. However, with an improved understanding of effects we could modularize this process. Rather than have each optimization be aware of exactly which effects are present, many optimizations could be modularized so that they need only be aware of what *properties* hold of the effects that are present. There would still be a need for effect analyses, but rather than report precisely what effects are present the analyses would only report what properties are present. This might enable optimization frameworks to be even more language independent.

To give an example of such a modular optimization, we consider common-subexpression elimination which might replace $e + e$, evaluating e twice, with essentially $+(\Delta(e))$, evaluating e once and using the value twice, provided this preserves semantics. Suppose the effect analysis determines that e has effect ε. We have already seen the property that must hold of ε for this transformation to be valid. The transformation replaces two evaluations of the same expression with a single evaluation whose value is used twice. This replacement is valid precisely for effects which laxly preserve diagonals, the same property we used for type generalization. Thus we can apply common-subexpression elimination when e may throw exceptions, may not terminate, is flexibly non-deterministic, only reads from the heap, or even only writes to the heap.

Many optimizations can be phrased in terms of abstract properties of effects. We give one more example: dead-code elimination. Given an insertable ε, we can apply dead-code elimination provided this effect *colaxly preserves terminals*. The terminal function $! : \tau \to 1$ (where 1 is the singleton type) is the unique effectless function from any type to the singleton type. An insertable effect ε colaxly preserves terminals when the top path is less flexible than the bottom path in the following diagram:



The idea is that, if a value is unused, then we can safely apply ! (or any effectless function) to the value. Thus if an effectful expression is unused then we can safely apply $map_\varepsilon(!)$ to it, essentially forgetting the value while propagating the effect. This corresponds to the bottom path. The top path corresponds to dropping the effectful expression altogether and replacing it with the singleton inhabitant of 1 with a trivial form of effect ε. This essentially corresponds to replacing a line of code with a skip. Thus if the top path is less flexible than the bottom path then we can replace any line of code producing an unused value with a skip (i.e. dead-code elimination).

Theorem. *Dead-code elimination can be applied to any dead code with an insertable effect that colaxly preserves terminals.*

By improving our understanding of effects we can improve our understanding of optimizations and build more modular compilers. Although here we have focused on small local optimizations, next we will show how the runtime system can apply our understanding of effects to large-scale program improvements.

6.3 Thread Parallelism

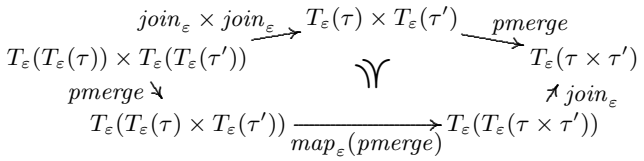
Parallelism is becoming exceedingly important in modern programming languages. As such, it is also important to understand the semantics of parallelism. Here we identify properties of effects for which steps of effectful computations in different threads can be interleaved arbitrarily, a key prerequisite to parallelism.

There is already some research along this line, specifically *commutative* monads [8]. A commutative monad is a monad in which left-to-right evaluation ($ltor$) is equivalent to right-to-left evaluation ($rtol$). This allows two lines of code to be reordered provided the value produced by the first line is not needed by the second line (i.e. they are value-independent). Thus even within a thread value-independent computations can be reordered arbitrarily, and so computations in separate threads can be interleaved arbitrarily.

However, we have determined that commutativity is too strong a requirement; there are many parallelizable effects which are not commutative. The key reason is that commutativity makes no distinction between computations in the same thread and computations in separate threads. Consider an effect for logging. It is important that logs in the same thread be printed in order; thus the logging effect is not commutative. However, logs in *separate* threads can be interleaved arbitrarily. This semantics does not correspond to either running thread 1 then thread 2 nor the other way around, so an effect can have a separate non-sequential semantics for multiple threads. In order to parallelize separate threads the semantics must allow the computations in these threads to be interleaved arbitrarily. Here we formalize properties which guarantee this requirement.

For simplicity, assume we have only one effect ε represented by a monad (it is straightforward to extend the properties we present below to a full effect system). Say we have two threads of lines of code, denoted by $lines \parallel lines'$, where each line has effect ε. Suppose we have a merge operation $pmerge$ formalizing execution of both threads. Our first property is that $pmerge$ must laxly preserve

the join operation of the monad. This means that the bottom path must be less flexible than the top path in the following diagram:

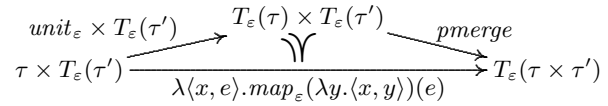


This essentially makes the following replacement valid:

$$\{line_1;line_2\} \parallel \{line_1';line_2'\} \;\Rightarrow\; (line_1 \parallel line_1');(line_2 \parallel line_2')$$

Thus two threads of sequences can be replaced with a sequence of two threads. Intuitively, threads can be run together step by step.

Our other requirement is that we can discard a thread that essentially does nothing. This is formalized by requiring the bottom path to be less flexible than the top path in the following diagram:



This must also hold should we swap left and right. The unit operation of a monad essentially corresponds to a skip line, so these requirements make the following replacements valid:

$$(skip \parallel line) \Rightarrow line \text{ and } (line \parallel skip) \Rightarrow line$$

Note that a monad is commutative precisely when the above properties use equality rather than flexibility; thus we are applying our new flexibility to generalize commutative monads.

Theorem. *If threads of sequential computations with effect ε are combined with a merge operation which laxly preserves joins and disregards units (formalized above), then the computations in these threads can be interleaved arbitrarily as required by parallelism.*

In particular, we can prove that left-to-right evaluation is a valid replacement of parallel execution of threads:

$$(line \parallel line') = \left\{\begin{matrix} line; \\ skip \end{matrix}\right\} \Bigg\| \left\{\begin{matrix} skip; \\ line' \end{matrix}\right\} \Rightarrow \begin{matrix} (line \parallel skip); \\ (skip \parallel line') \end{matrix} \Rightarrow \left\{\begin{matrix} line; \\ line' \end{matrix}\right\}$$

Here we also use the identity laws of monadic units. If we furthermore use the associativity law of monadic joins, then we can use similar techniques to replace parallel threads with any interleaving.

Parallelism is only one runtime-systems application of our improved understanding of effects. This understanding can also be used to classify when computations can be safely shipped over a faulty network, when results can be cached, or when evaluation can be speculative or lazy. Thus, as with compiler optimizations, there is opportunity to modularize runtime optimizations and effects, enabling language-independent tools for runtime systems.

7. Conclusion

We have presented a framework for formalizing the denotational semantics of effects. Not only does our framework generalize traditional monadic techniques, but it classifies precisely which effects systems can be formalized using traditional techniques. Furthermore, our framework formalizes the semantics of many roles of effects which have previously disregarded for the most part. Our framework separates effect systems into a nominal component, for naming effects, and a semantic component, for giving denotation semantics to effects. This separation makes it easy to express properties of effect systems in a language-independent manner. In particular, we have shown that our framework can be applied to many stages in language design and implementation in order to produce language-independent tools which can be used whenever their abstractly specified requirements are satisfied. Thus, by improving the understanding of effects our framework helps one formally reason about languages at a higher level.

This paper is truly only an introduction to semantic effect systems. For example, we made many simplifications for sake of explanation such as merging the type domain and semantic domain, and using normal category theory rather than indexed/fibred-category theory which is used to formalize polymorphic and dependent types and effects. While here we have focused on introduction and interaction of effects, one can extend our techniques to coherent elimination of effects, such as with try-catch blocks. In the future, we hope to find a way to combine denotational and operational semantics so that language designers and implementers may enjoy the benefits of both worlds. Despite the abbreviates we made, we believe the framework as presented here illustrates the key concepts for properly understanding effects in such a way that enables theorems and tools to be specified in a language-independent manner.

References

[1] J. Adámek, H. Herrlich, and G. E. Strecker. *Abstract and Concrete Categories*. Wiley-Interscience, 1990.

[2] N. Benton, J. Hughes, and E. Moggi. Monads and effects. In *International Summer School on Applied Semantics*, 2000.

[3] J. Garrigue. Relaxing the value restriction. In *FLOPS*, 2003.

[4] R. Godement. *Topologie Algébrique et Théorie des Faisceaux*. Hermann, 1958.

[5] B. Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, 1999.

[6] P. T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium*, volume 1. Oxford University Press, 2002.

[7] M. P. Jones and L. Duponcheel. Composing monads. Technical report, Yale University, New Haven, CT, USA, Dec. 1993.

[8] M. P. Jones and P. Hudak. Implicit and explicit parallel programming in Haskell. Technical report, Yale University, New Haven, CT, USA, Aug. 1993.

[9] S. P. Jones and P. Wadler. Imperative functional programming. In *POPL*, 1993.

[10] R. B. Kieburtz. Taming effects with monadic typing. In *ICFP*, 1998.

[11] D. J. King and P. Wadler. Combining monads. In *ETAPS*, 1992.

[12] H. Kleisli. Every standard construction is induced by a pair of adjoint functors. *Proceedings of the American Mathematical Society*, 16(3):544–546, 1965.

[13] T. Leinster. *Higher Operads, Higher Categories*. Cambridge University Press, 2004.

[14] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL*, 1995.

[15] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, 1988.

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

[17] D. Marino and T. Millstein. A generic type-and-effect system. In *TLDI*, 2009.

[18] E. Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-90-113, Edinburgh University, 1988.

[19] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

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

[21] R. Street. Two constructions on lax functors. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 13(3):217–264, 1972.

[22] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2:245–271, 1992.

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

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

[25] P. Wadler. Comprehending monads. In *LISP and Functional Programming*, 1990.

[26] P. Wadler. The essence of functional programming. In *POPL*, 1992.

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

[28] A. K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.