A Flexible Semantic Framework for Effects

Ross Tate[1] and Daan Leijen[2]

[1] University of California, San Diego
[2] Microsoft Research, Redmond

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 non-deterministically 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 abstractly, 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 (the space in which type-and-effect analyses [14, 16, 19, 22, 23, 25] work) and a *semantic* component for defining the *denotational semantics* of effects (the space in which monadic techniques typically work). Our framework is also general, making only the assumptions necessary for semantics to be *coherent*, meaning 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 techniques for formalizing the semantics of effects, namely monadic techniques [2, 9, 17, 18, 26]. 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 [28]. 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 also 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. 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. We show that the value restriction [29] can be relaxed in the presence of effects satisfying an abstract property phrased in terms of our framework. We also 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.

To summarize, this paper makes the following contributions:

Section 2 A definition of nominal effect systems for naming effects, a specific class of which includes the many type-and-effect systems used in analyses
Section 3 A framework for formalizing the denotational semantics of effect systems
Section 4 A classification of the nominal effect systems whose semantics can be formalized using monads, and examples of systems for which this is not possible
Section 5 Extensions for effect systems with lateral composition and flexible semantics
Section 6 Applications of our framework to type generalization and to parallelism

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, i.e. systems for *naming* effects. In Section 3, we present *semantic* effect systems, i.e. systems which give semantics to nominal effect systems. In these two sections, we focus on nominal and semantic effect systems for only *sequential* programs of the form

$$\{x \leftarrow e;\ \ y \leftarrow e';\ \ \mathbf{return}\, e''\}$$

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 languages, such as those in which subexpressions as well as lines of code have effects. 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 are also dynamic nominal effect systems particularly useful for defining the semantics of a language.

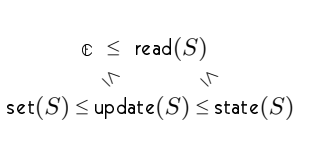
Nominal effect systems, like type systems, often have a notion of subeffects. A procedure with nominal effect ε also has nominal effect ε' when ε is a subeffect of ε'. As with types, a subeffect is typically more precise or restrictive than a supereffect. Thus nominal effect systems are essentially (static 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 updates the heap, then the combination reads-and-updates the heap.

We formalize nominal effect systems using the following definition.

Definition 1. *A nominal effect system is a set* EFF *of nominal effects (i.e. effect names) and a distinguished basic effect* $\mathbb{e}$ *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* $\mathbb{e}$ *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:*

$$\varepsilon_1 \leq \varepsilon_1' \wedge \varepsilon_2 \leq \varepsilon_2' \implies \varepsilon_1 \fatsemi \varepsilon_2 \leq \varepsilon_1' \fatsemi \varepsilon_2'$$



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

Fig. 1. Nominal Effect System for State Machines

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 [16]. 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, 14, 25]. In particular, there is usually a set of primitive effects, and EFF is simply all finite sets of primitive effects [2, 16, 19, 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.

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 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:

$$\textsc{Eff} = \{\mathsf{nd}(n) \mid n \in \mathbb{N} \wedge n \geq 1\} \qquad \mathsf{nd}(m) \leq \mathsf{nd}(n) = m \leq n$$
$$\mathbb{c} = \mathsf{nd}(1) \qquad \mathsf{nd}(m) \fatsemi \mathsf{nd}(n) = \mathsf{nd}(m * n)$$

Fig. 2. Nominal Effect System for Bounded Non-Determinism

$$\underset{\text{(gets the current state)}}{get : \mathtt{unit} \xrightarrow{\mathsf{read}(S)} S} \qquad \underset{\text{(sets the current state)}}{put : S \xrightarrow{\mathsf{set}(S)} \mathtt{unit}} \qquad \underset{\text{(updates the current state)}}{modify : (S \xrightarrow{\mathbb{c}} S) \xrightarrow{\mathsf{update}(S)} \mathtt{unit}}$$

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 Section 3 we will use denotational semantics to prove that it is sound. We should also note that Haskell has the same get, put, and $modify$ in its library, but 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 $arb_n : \tau^n \xrightarrow{\mathsf{nd}(n)} \tau$ for each positive integer n. 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:

$$\{x \leftarrow arb_2(1,5);\ \ y \leftarrow arb_2(x+1, x-1);\ \ \mathbf{return}\ y * 2\}$$

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 formal semantics is much simpler.

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; the technical report [24] provides additional characterizations. In Section 3, 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 $\langle \textsc{Eff}, \mathbb{c}, \fatsemi \rangle$ 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 by 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 or can be made insertable except for one: the $\mathtt{set}(S)$ 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.

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). So although sequentially compressive effects are common, they are by no means necessary.

Increasing Systems 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 $\mathtt{set}(S)$ followed by $\mathtt{read}(S)$ is $\mathtt{set}(S)$, which is *not* a supereffect of $\mathtt{read}(S)$, rather than $\mathtt{state}(S)$, which is a supereffect both. However, we believe that *not* requiring sequential composition to be increasing is a powerful aspect of our framework because it enables effects to interact in a way *besides* containment, discussed more in Section 4. 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 $\mathtt{set}(S)$ effect, put as having the $\mathtt{read}(S)$ effect, and $\mathtt{read}(S) \fatsemi \mathtt{read}(S)$ 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 computational 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 [17]. In 1990, Wadler carried this concept over to the functional languages community [26, 27], 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: $\mathbb{e} = \mathsf{partial}$. 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 [17] 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 to define 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, in this case $\lambda d.d + 1$. We can do this by using a map operation:

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

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

Fig. 3. Semantic Effect System for Bounded Non-Determinism

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 $T : \textsc{Type} \to \textsc{Type}$ 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 [1]. 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 2. *A semantic effect system for a nominal effect system* $\langle \textsc{Eff}, \mathfrak{e} \rangle$ *specifies for each effect* ε *in* $\textsc{Eff}$ *a functor* $\langle T_\varepsilon, map_\varepsilon \rangle$ *indicating how to describe the effect and propagate it through computations, and also specifies an operation* $unit : \tau \to T_{\mathfrak{e}}(\tau)$ *(specifically a natural transformation [1]) 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 $\mathfrak{e}$. This operation is related to monadic units, but it is required *only* for the basic effect $\mathfrak{e}$. For our running example with only the single effect $\mathsf{partial}$, the $unit$ operation is defined as follows:

$$unit : \tau \to \text{Partial}(\tau)$$
$$unit(x) = \text{success}(x)$$

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 non-deterministically producing only that constant.

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 the semantics of this expression as $map_{\mathsf{partial}}(\lambda d.d \div y)(64 \div x)$. 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:

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

Essentially $join_{\mathsf{partial}}$ 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 3. *A sequential semantic effect system for a sequential nominal effect system* $\langle \text{EFF}, \mathfrak{C}, \fatsemi \rangle$ *additionally specifies for each pair of effects* ε *and* ε' *a join operation 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 equational requirements, described in the technical report [24], that are necessary and sufficient for the semantics to be ***coherent****, meaning all possible ways of inserting implicit operations lead to equivalent semantics.*

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 $join_{\mathsf{nd}(m),\mathsf{nd}(n)}$ 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, for $n \geq 2$, it is impossible to define a natural transformation 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 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:

$$\begin{array}{ll} x \leftarrow e; & \text{(has effect } \varepsilon\text{)} \\ y \leftarrow f(x); & \text{(has no effect)} \\ \texttt{return } g(x,y) & \text{(has effect } \varepsilon'\text{)} \end{array}$$

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 and which variables the expressions f and g actually use. All of these translations are equivalent, though, due to the equational requirements of our framework, even though multiple effects are present. We following is the most naïve of these translations.

$$join_{\varepsilon,\mathbb{c}\,\S\,\varepsilon'}(map_{\varepsilon}(\lambda x.join_{\mathbb{c},\varepsilon'}(map_{\mathbb{c}}(\lambda y.g(x,y))(unit(f(x)))))(e))$$

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

Definition 4. *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 equational requirements, described in our technical report [24], that are necessary and sufficient for the semantics to be coherent.

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, demonstrating that the nominal sequential composition we defined for this effect system is in fact a sound approximation. Demonstrating soundness of an abstraction is one use of denotational semantics, but next we see how it can allow one to classify and analyze nominal and semantic 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.

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: $\varepsilon \xrightarrow{convert_{\varepsilon \le \varepsilon'}} \varepsilon'$

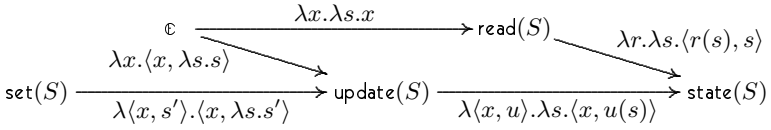


Table of Join Operations

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

the remaining $join$ operations are uniquely determined by effect containment (see Section 4)

Fig. 4. Semantic Effect System for State Machines

Tightness We define a notion of *tightness*, which provides a way to evaluate the precision of effect systems. We say an operation in a nominal effect system is *tight* if it has a semantic effect system such that the corresponding semantic operation is surjective in some sense. For example, the primitive operation $put : S \xrightarrow{\mathsf{set}(S)} \mathtt{unit}$ is tight because each element in $T_{\mathsf{set}(S)}(\mathtt{unit})$ can directly result from a use of put. However, Haskell gives put the $\mathsf{state}(S)$ effect, but no non-constant semantic function in $T_{\mathsf{state}(S)}(\mathtt{unit}) = S \to \mathtt{unit} \times S$ can directly arise from a use of put, so put is not tight/surjective in this system. In fact, *there is no tight sound monadic formalization of* put, so our framework is strictly necessary for more precise abstractions of effects.

As for subeffects, a subeffect system has tight joins if, whenever ε and ε' have a join $\varepsilon \sqcup \varepsilon'$ in the subeffect preorder, the pair of operations $\langle convert_{\varepsilon \le \varepsilon \sqcup \varepsilon'}, convert_{\varepsilon' \le \varepsilon \sqcup \varepsilon'}\rangle$ 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 tight joins, then there is 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 *or* just set the state.

Sequential composition is tight if each $join$ operator is surjective (specifically an *extremal epimorphism* [1]), so 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. However, 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, illustrating the precision we enable by *not* requiring sequential composition to be increasing.

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 $\mathsf{state}(S)$ effect is lossless. 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 $\mathsf{exc}(E)$ and non-deterministic exceptions $\mathsf{nd\text{-}exc}(E)$ (where E is the set of possible exceptions). It is quite reasonable to have $\mathsf{exc}(E)$ be a subeffect of $\mathsf{nd\text{-}exc}(E)$. However, this conversion is lossy, as arbitrary non-deterministic processes cannot be made deterministic in a way that undoes the conversion. Nonetheless, it is good to take losslessness into consideration when reasoning about effect system.

3.3 Category-Theoretic Formalization

Coincidentally, our various operations and equational requirements for nominal and semantic effect systems form what is known as a lax functor [6, 21] from a one-object locally thin 2-category to the 2-category of categories. The 2-category of effectful computations for an effect system turns out to be what is known as the lax colimit [6, 21] of this lax functor. This demonstrates that our framework is a natural formalization of effects, neither being too relaxed nor too restrictive. The structure of a lax functor also provides ways to compare effect systems. However, we leave discussion of these higher-level concepts to our technical report [24].

4 Monadic Semantics

Monads have been the traditional approach for formalizing the denotational 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, $unit$, and $join$ 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 actually be a benefit. In Section 5, we will present extensions to our framework for handling additional roles effects play in a language that 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 [28], 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 nondeterminism effects have no monadic join, and the $\mathtt{set}(S)$ 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 1. *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. [24]*

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 [12] or a monad morphism in the functional languages community [2, 17, 26]. This fact, combined with the above lemma, entails the following corollary.

Corollary 1. *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 a trademark of these systems. Recognizing this, we can use Corollary 1 to prove that Wadler and Thiemann's claimed marriage between effects and monads holds particularly for *analytical* effect systems.

Theorem 1. *The denotational semantics of any analytical effect system is formalized by a hierarchy of monads and monad morphisms because they use the join operator* $\sqcup$ *on their join semi-lattice of nominal effects for nominal sequential composition* $\fatsemi$*. [24]*

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 denotational *semantics* of analytical effect systems are always formalized using monads. However, the semantic effect systems in Figures 3 and 4 are not monads. This is because the bounded nondeterminism nominal effects are not sequentially compressive, and the $\mathtt{set}(S)$ nominal effect is not insertable. In particular, $\varepsilon_1 \fatsemi \varepsilon_2$ is *not* $\varepsilon_1 \sqcup \varepsilon_2$ for these systems.

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 analytical effect systems, nominal sequential composition is always increasing and sequentially compressive due to the nature of joins on preorders. Because of this, the interactions of distinct effects is completely determined by their containment in some larger effect, as formalized by the following theorem.

Theorem 2. *Semantic sequential composition for different effects in increasing sequentially compressive nominal effect systems is uniquely determined by the convert operations and semantic sequential composition of effects with themselves. [24]*

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

This means that interaction of distinct effects in these systems is very restricted. In fact, the multitude of monadic techniques for building effect systems all handle interaction of distinct effects by packing their monads into one larger monad [5, 7, 11, 13, 15]. 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 between distinct effects, such as that for $\mathsf{read}(S)$ after $\mathsf{set}(S)$. 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. The technical report contains an additional extension identifying the additional structure necessary for effects to be used in infinite processes such as unbounded loops and recursion [24]. These additions make our framework capable of formalizing the semantics of effects in a large class of imperative and functional languages. In Section 6, 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 5. *Nominal lateral composition specifies an associative binary operator* $\fatsemi : \textsc{Eff} \times \textsc{Eff} \to \textsc{Eff}$ *with* $\mathbb{c}$ *as the identity element. For typed nominal effect systems,* $\fatsemi$ *can be a partial operator. Should a nominal effect system have both subeffects and lateral composition, then lateral composition must preserve subeffects.*

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

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

This family of merge operations must satisfy equational requirements, described in the technical report [24], that are necessary and sufficient for the semantics to be ***coherent****.*

In the same way semantic sequential composition generalizes monads from one effect to multiple effects, semantic lateral composition generalizes *lax monoidal functors* [12] from one effect to multiple effects.

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 reversed 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 obvious merge operations: given a pair of vectors simply construct a vector of pairs componentwise, and 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 $5 * \boldsymbol{x} + 10 * \boldsymbol{y} - 1$. 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 $\mathsf{exc}(E)\,,\,\mathsf{exc}(E)$ 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 essentially specifies a preorder $\preccurlyeq_\varepsilon$ on $T_\varepsilon(\tau)$; the technical report contains the details [24]. $e \preccurlyeq_\varepsilon e'$ 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 see 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.

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 and to formalize parallelism in the presence of effects. Applications to optimization can be found in the technical report [24].

6.1 Type Generalization

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

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

If we used 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 [29] 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 specifically $T_\varepsilon(\forall\alpha.\alpha \to \alpha)$, 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 discusses in the technical report [24]. 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 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:

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

That is, making two copies of an effectful expression and then evaluating both copies (top path) is equivalent to evaluating once and duplicating the value (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 3. *In a language with a common form of parametric polymorphism, type generalization is always sound for expressions with a laterally compressive effect that laxly preserves diagonals. [24]*

Thus type generalization is always sound when the expression just might throw exceptions, might not terminate, is flexibly non-deterministic, only reads from the heap, or even only writes to the heap.

6.2 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, 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 (formalized in the technical report [24]). 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 easy to extend the properties 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 first diagram of Figure 5.

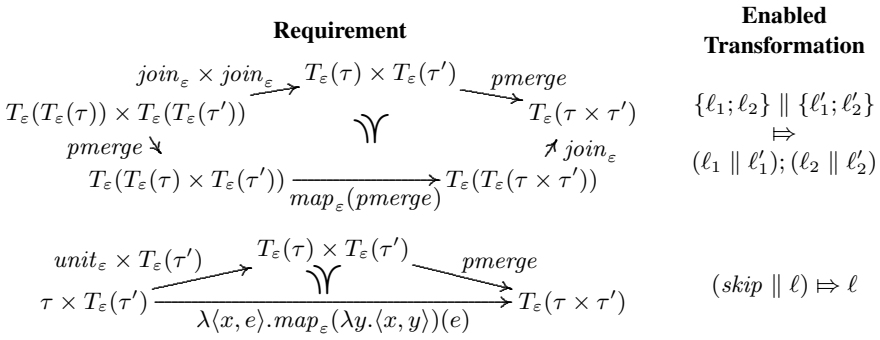


Fig. 5. Necessary requirements for parallelizing effectful computations

This requirement essentially means 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 second diagram of Figure 5. This must also hold should we swap left and right. This requirement and its swapped version essentially allow us to remove empty code running in parallel with other code. Note that a monad is commutative precisely when the requirements in Figure 5 use equality rather than flexibility; thus we are applying our new concept of flexibility to generalize commutative monads.

Theorem 4. *If threads of sequential computations with effect ε are combined with a merge operation which laxly preserves joins and disregards units (Figure 5), then the computations in these threads can be interleaved arbitrarily as required by parallelism.*

For example, we can easily show that left-to-right execution is valid for parallel threads:

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

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 been disregarded for the most part. Our framework separates effect systems into a nominal component, for naming effects, and a semantic component, for giving denotational semantics to effects. This separation makes it easy to express properties of effect systems abstractly, and we have shown that the requirements of many stages in language design and implementation can be stated in the abstract terminology enabled by our framework. Thus, by improving the understanding of effects our framework helps one formally reason about languages at a higher level.

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. M. Hyland, G. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1):70–99, 2006.
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. T. Leinster. *Higher Operads, Higher Categories*. Cambridge University Press, 2004.
13. S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL*, 1995.
14. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, 1988.
15. C. Lüth and N. Ghani. Composing monads using coproducts. *ACM SIGPLAN Notices*, 37(9):133–144, 2002.
16. D. Marino and T. Millstein. A generic type-and-effect system. In *TLDI*, 2009.
17. E. Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-90-113, Edinburgh University, 1988.
18. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
19. F. Nielson and H. R. Nielson. Type and effect systems. In *ACM Computing Surveys*, 1999.
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. R. Tate, D. Leijen, and S. Lerner. A flexible semantic framework for effects. Technical report, University of California, San Diego and Microsoft Research, Redmond, Available at `http://cseweb.ucsd.edu/~rtate/effectstr.pdf`, 2010.
25. A. P. Tolmach. Optimizing ML using a hierarchy of monadic types. In *Types in Compilation*, 1998.
26. P. Wadler. Comprehending monads. In *LISP and Functional Programming*, 1990.
27. P. Wadler. The essence of functional programming. In *POPL*, 1992.
28. P. Wadler and P. Thiemann. The marriage of effects and monads. *Transactions on Computational Logic*, 4(1):1–32, 2003.
29. A. K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.