A Generic Approach to Flow-Sensitive Polymorphic Effects

Colin S. Gordon

Drexel University
csgordon@drexel.edu

Abstract

Effect systems are lightweight extensions to type systems that can verify a wide range of important properties with modest developer burden. But our general understanding of effect systems is limited primarily to systems where the order of effects is irrelevant. Understanding such systems in terms of a lattice of effects grounds understanding of the essential issues, and provides guidance when designing new effect systems. By contrast, sequential effect systems — where the order of effects is important — lack a clear algebraic characterization.

We derive an algebraic characterization from the shape of prior concrete sequential effect systems. We present an abstract polymorphic effect system with singleton effects parameterized by an effect quantale — an algebraic structure with well-defined properties that can model a range of existing order-sensitive effect systems. We define effect quantales, derive useful properties, and show how they cleanly model a variety of known sequential effect systems. We show that effect quantales provide a free, general notion of iterating a sequential effect, and that for systems we consider the derived iteration agrees with the manually designed iteration operators in prior work. Identifying and applying the right algebraic structure led us to subtle insights into the design of order-sensitive effect systems, which provides guidance on non-obvious points of designing order-sensitive effect systems. Effect quantales have clear relationships to the recent category theoretic work on order-sensitive effect systems, but are explained without recourse to category theory. In addition, our derived iteration construct should generalize to these semantic structures, addressing limitations of that work.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases Type systems, effect systems, quantales, polymorphism

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.13

1 Introduction

Effect systems are a well-known lightweight extension to standard type systems, which are capable of verifying an array of useful program properties with modest developer effort. They have proven useful for enforcing error handling [59, 4, 29], ensuring a variety of safety properties for concurrent programs [18, 19, 17, 10, 9, 20], purity [33, 16], safe arena-based memory management [41, 55, 57], and more. Effect systems extend type systems to track not only the shape of and constraints on data, but also a summary of the side effects caused by an expression's evaluation. Java's checked exceptions are the best-known example of an effect system — the effect of an expression is the set of (checked) exceptions it may throw — and other effects have a similar flavor, like the set of heap regions accessed by parallel code, or the set of locks that must be held to run an expression without data races.

However, our understanding of effect systems is concentrated in the space of systems like Java's checked exceptions, where the *order of effects* is irrelevant: the system does not care that an `IllegalArgumentException` would be thrown before any possible `IOException`.

31st European Conference on Object-Oriented Programming (ECOOP 2017).
Editor: Peter Müller; Article No. 13; pp. 13:1–13:31
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Effects in such systems are characterized by a join semilattice, which captures exactly systems where ordering is irrelevant (since the join operation is commutative and associative). This is an impressively large and useful class of systems, but the assumption that order is irrelevant leaves some of the more sophisticated effect systems for checking more powerful properties out of reach. We refer to this class of effect systems — the traditional default — as *commutative* effect systems, to contrast against the class we study in this paper. The alternative class of effect systems, where the order in which effects occur matters — *sequential* effect systems, following Tate's terminology [56][1] — reason directly about the proper ordering of program events. Examples include non-block-structured reasoning about synchronization for data races and deadlock freedom [9, 28, 53], atomicity [21, 20], and memory management [11].

Effect system design for the traditional commutative effect systems has been greatly aided in both theory and practice by the recognition that effects in such systems form a bounded join semilattice — a lattice with top, bottom, and all binary joins (least-upper-bounds). On the theory side, this permits general formulations of effect systems to study common properties [42, 50, 3]. On the practical side, this guides the design and implementation of working effect systems. If an effect system is not a join semilattice, why not? (Usually this indicates a mistake.) Effect system frameworks can be implemented generically with respect to an effect lattice [50, 58], and in the common case where effects are viewed as sets of required capabilities, simply specifying the capabilities and exploiting the default powerset lattice makes core design choices straightforward. In the research literature, the ubiquity of lattice-based (commutative) effect systems simplifies explanations and presentations.

Sequential effect systems so far have no such established common basis in terms of an algebraic structure to guide design, implementation, and comparison, making all of these tasks more difficult. Recent work on semantic approaches to modeling sequential effect systems [56, 36, 45] has produced very general characterizations of the mathematics behind key *necessary* constructs (namely, sequencing effects), but with one recent exception [45] does not produce a description that is *sufficient* to model a complete sequential effect system for a real language. Partly this stems from the fact that the accounts of such work proceed primarily by generalizing categorical structures used to model sequential computation, rather than implementing complete source-level effect systems. None of this work has directly considered effect polymorphism (essential for any real use), singleton effects (required for prominent effect systems both commutative and sequential), or iteration constructs. So there is currently a gap between this powerful semantic work, and understanding real sequential effect systems in a systematic way.

We generalize directly from real source-level type-and-effect systems to produce an algebraic characterization for sequential effect systems, suitable for modeling some well-known sequential type-and-effect disciplines, and (we hope) useful for guiding the design of future sequential effect systems. We give important derived constructions (products, and inducing an iteration operation on effects), and put them to use with an explicit translation between Flanagan and Qadeer's early atomicity type system [21] and a (sequential) equivalent built as an instantiation of our generic sequential type-and-effect system.

Overall, our contributions include:

- A new algebraic characterization of sequential effect systems — effect quantales — that is consistent with existing semantic notions and easily subsumes commutative effects

[1] These effect systems have been alternately referred to as flow-sensitive [42], as they are often formalized using flow-sensitive type judgments (with pre- and post-effect) rather than effects in the traditional sense. However, this term suggests a greater degree of path sensitivity and awareness of branch conditions than most such systems have. We use Tate's terminology as it avoids technical quibbles.

- A syntactic motivation for effect quantales by generalizing from concrete, full-featured sequential effect systems. As a result, we are the first to investigate interplay between singleton effects and sequential effect systems in the abstract (not yet addressed by semantic work). This reveals subtlety in the metatheory of sequential effects that depend on program values.
- Demonstration that effect quantales are not only general, but also sufficient to modularly define the structure of existing non-trivial effect systems.
- A general characterization of effect iteration for any sequential effect system given by an effect quantale, including demonstration that the resulting iteration for prior systems (as effect quantales) exactly matches the hand-constructed iteration of the original work. The form is general enough that it should adapt to semantic characterizations as well.
- The first generic *sequential* effect system with effect polymorphism.
- Precise characterization of the relationship between effect quantales and related notions, ultimately connecting the syntax of established effect systems to semantic work, closing a gap in our understanding.

2 Background on Commutative and Sequential Effect Systems

Here we derive the basic form of a new algebraic characterization of sequential effects based on generalizing from the use of effects in current sequential effect systems. The details of this form are given in Section 3, with a corresponding generic type-and-effect system in Section 6. We refer to the two together as a framework for sequential effect systems.

By now, the standard mechanisms of commutative effect systems — what is typically meant by the phrase "type-and-effect system" — are well understood. The type judgment $\Gamma \vdash e : \tau$ of a language is augmented with a component χ describing the overall effect of the term at hand: $\Gamma \vdash e : \tau \mid \chi$. Type rules for composite expressions, such as forming a pair, join the effects of the child expressions by taking the least upper bound of those effects (with respect to the effect lattice). And the final essential adjustment is to handle the *latent effect* of a function — the effect of the function body, which is deferred until the function is invoked. Function types are extended to include this latent effect, and this latent effect is included in the effect of function application. Allocating a closure itself has no meaningful effect, and is typically given the bottom effect in the semilattice:

$$\textsc{T-Fun}\ \frac{\Gamma, x : \tau \vdash e : \tau' \mid \chi}{\Gamma \vdash (\lambda\, x.\, e) : \tau \xrightarrow{\chi} \tau' \mid \bot} \qquad \textsc{T-Call}\ \frac{\Gamma \vdash e_1 : \tau \xrightarrow{\chi} \tau' \mid \chi_1 \qquad \Gamma \vdash e_2 : \tau \mid \chi_2}{\Gamma \vdash e_1\ e_2 : \tau' \mid \chi_1 \sqcup \chi_2 \sqcup \chi}$$

Consider the interpretation for concrete effect systems. Java's checked exceptions are an effect system [29, 59]: the effects are sets of (checked) exceptions ordered by inclusion, with set union as the semilattice join. The `throws` clause of a method states its *latent* effect — the effect of actually *executing* the method (roughly χ in T-Fun above). The exceptions thrown by a composite expression such as invoking a method is the union of the exceptions thrown by subexpressions (e.g., the receiver object expression and method arguments) and the latent effect of the code being executed (as in T-Call above). Most effect systems for treating data race freedom (for block-structured synchronization like Java's `synchronized` blocks, such as RCC/Java [19, 1]) use sets of locks as effects, where an expression's effect is the set of locks guarding data that may be accessed by that expression. The latent effect there is the set of locks a method requires to be held by its call-site. Other effect systems follow similar structure: a binary yes/no effects of whether or not code performs a sensitive class of action like allocating memory in an interrupt handler [33, 34, 16] or accessing user interface elements [27]; tracking the sets of memory regions read, written, or allocated into for safe memory deallocation [55, 57] or parallelizing code safely [41, 25] or even deterministically [8, 37].

But these and many other examples do not care about ordering. Java does not care which exception might be thrown first. Race freedom effect systems for block-structured locking do not care about the order of object access within a `synchronized` block. Effect systems for region-based memory management do not care about the order in which regions are accessed, or the order of operations within a region. Because the order of combining effects in these systems is irrelevant, we refer to this style of effect system as *commutative* effect systems, though due to their prevalence and the fact that they arose first historically, this is the class of systems typically meant by general references to "effect systems."

Sequential effect systems tend to have slightly different proof theory. Many of the same issues arise (latent effects, etc.) but the desire to enforce a sensible *ordering* among expressions leads to slightly richer type judgments. Often they take the form $\Gamma;\ \Delta \vdash e : \tau \mid \chi \dashv \Delta'$. Here the Δ and Δ' are some kind of pre- and post-state information — for example, the sets of locks held before and after executing e [53], or abstractions of heap shape before and after e's execution [28]. χ as before is an element of some lattice, such as Flanagan and Qadeer's atomicity lattice (Figure 1). Some sequential effect systems have both of these features, and some only one or the other. (These components never affect the type of variables, and strictly reflect some property of the *computation* performed by e, making them part of the effect.) The judgments for something like a variant of Flanagan and Qadeer's atomicity type system that tracks lock sets flow-sensitively rather than using synchronized blocks or for an effect system that tracks partial heap shapes before and after updates [28] might look like the following, using Δ or Υ to track locks held, and tracking atomicities with χ:

$$\frac{\Gamma, x : \tau;\ \Upsilon \vdash e : \tau' \mid \chi \dashv \Upsilon'}{\Gamma;\ \Delta \vdash (\lambda\, x.\, e) : \tau \xrightarrow{\Upsilon,\chi,\Upsilon'} \tau' \mid \bot \dashv \Delta} \qquad \frac{\Gamma;\ \Delta \vdash e_1 : \tau \xrightarrow{\Delta'',\chi,\Delta'''} \tau' \mid \chi_1 \dashv \Delta' \quad\ \Gamma;\ \Delta' \vdash e_2 : \tau \mid \chi_2 \dashv \Delta''}{\Gamma\Delta \vdash e_1\ e_2 : \tau' \mid \chi_1;\ \chi_2;\ \chi \dashv \Delta'''}$$

The sensitivity to evaluation order is reflected in the threading of Δs through the type rule for application, as well as through the switch to the sequencing composition ; of the basic effects. Confusingly, while χ continues to be referred to as the effect of this judgment, the real effect is actually a combination of χ, Δ, and Δ' in the judgment form. This distribution of the "stateful" aspects of the effect through a separate part of the judgment obscures that this judgment really tracks a product of *two* effects — one concerned with the self-contained χ, and the other a form of effect indexed by pre- and post-computation information.

Rewriting these traditional sequential effect judgments in a form closer to the commutative form reveals some subtleties of sequential effect systems:

$$\frac{\Gamma, x : \tau \vdash e : \tau' \mid (\Upsilon \rightsquigarrow \Upsilon') \otimes \chi}{\Gamma \vdash (\lambda\, x.\, e) : \tau \xrightarrow{(\Upsilon \rightsquigarrow \Upsilon') \otimes \chi} \tau' \mid (\Delta \rightsquigarrow \Delta) \otimes \bot}$$

$$\frac{\Gamma \vdash e_1 : \tau \xrightarrow{(\Delta'' \rightsquigarrow \Delta''') \otimes \chi} \tau' \mid (\Delta \rightsquigarrow \Delta') \otimes \chi_1 \qquad \Gamma \vdash e_2 : \tau \mid (\Delta' \rightsquigarrow \Delta'') \otimes \chi_2}{\Gamma \vdash e_1\ e_2 : \tau' \mid ((\Delta \rightsquigarrow \Delta');\ (\Delta' \rightsquigarrow \Delta'');\ (\Delta'' \rightsquigarrow \Delta''')) \otimes (\chi_1;\ \chi_2;\ \chi)}$$

One change that stands out is that the effect of allocating a closure is not simply the bottom effect (or product of bottom effects) in some lattice. No sensible lattice of pre/post-state pairs has equal pairs as its bottom. However, it makes sense that some such equal pair acts as the left and right identity for *sequential* composition of these "stateful" effects. In commutative effect systems, sequential composition is actually least-upper-bound, for which the identity element happens to be $\bot$. We account for this in our framework.

We also assumed, in rewriting these rules, that it was sensible to run two effect systems "in parallel" in the same type judgment, essentially by building a product of two effect systems. Some sequential effect systems are in fact built this way, as two "parallel" systems (e.g., one for tracking locks, one for tracking atomicities, one for tracking heap shapes, etc.) that together ensure the desired properties. The general framework we propose supports a straightforward product construction.

Another implicit assumption in the refactoring above is that the effect tracking that is typically done via flow-sensitive type judgments is equivalent to *some* algebraic treatment of effects akin to how χs are managed above. While it is clear we would *want* a clean algebraic characterization of such effects, the existence of such an algebra that is adequate for modeling known sequential effect systems for non-trivial languages is not obvious. Our proposed algebraic structures (Section 3) are adequate to model such effects (Section 4).

Examining the sequential variant of other rules reveals more subtleties of sequential effect system design. For example, effect joins are still required in sequential systems:

$$\frac{\Gamma\vdash e:\mathcal{B}\mid\chi\quad\Gamma\vdash e_1:\tau\mid\chi_1\quad\Gamma\vdash e_2:\tau\mid\chi_2}{\Gamma\vdash \mathsf{if}\ e\ e_1\ e_2:\tau\mid\chi\sqcup\chi_1\sqcup\chi_2}\Rightarrow\frac{\Gamma\vdash e:\mathcal{B}\mid\chi\quad\Gamma\vdash e_1:\tau\mid\chi_1\quad\Gamma\vdash e_2:\tau\mid\chi_2}{\Gamma\vdash \mathsf{if}\ e\ e_1\ e_2:\tau\mid\chi;\ (\chi_1\sqcup\chi_2)}$$

Nesting conditionals can quickly produce an effect that becomes a mass of alternating effect sequencing and join operations. For a monomorphic effect system, concrete effects can always be plugged in and comparisons made. However, for a polymorphic effect system, it is highly desirable to have a sensible way to simplify such effect expressions — particularly for highly polymorphic code — to avoid embedding the full structure of code in the effect. Our proposal codifies natural rules for such simplifications.

3 Effect Quantales

Quantales [43, 44] are an algebraic structure originally proposed to generalize some concepts in topology to the non-commutative case, which later found use in models for non-commutative linear logic [61] and reasoning about observations of computational processes [2], among other uses. Abramsky and Vickers give a thorough historical account [2]. They are almost what is required for modeling sequential effect systems, but carry a bit too much structure, so we define here a slightly less constrained variant called *effect quantales*. We establish one very useful property of effect quantales, and show how they subsume commutative effect systems. We defer more involved examples to Section 4.

▶ **Definition 1** (Effect Quantale). An *effect quantale* $Q=(E,\sqcup,\rhd,\top,I)$ is a join semilattice $(E,\sqcup)$ with top element $\top$ with monoid $(E,\rhd,I)$, such that $\rhd$ distributes over joins in both directions — $a\rhd(b\sqcup c)=(a\rhd b)\sqcup(a\rhd c)$ and $(a\sqcup b)\rhd c=(a\rhd c)\sqcup(b\rhd c)$ — and $\top$ is a nilpotent element for the monoid ($a\rhd\top=\top=\top\rhd a$).

As is standard in lattice theory, we induce the partial order $x\sqsubseteq y\overset{\mathsf{def}}{=}x\sqcup y=y$ from the join operation, which ensures the properties required of a partial order.

We will use the semilattice to model the standard effect hierarchy, using the partial order for subeffecting. The (non-commutative) monoid operation $\rhd$ will act as the sequential composition. The properties of the semilattice and distributivity of the product over joins will permit us to move common prefixes or suffixes of effect sequences into or out of least-upper-bounds of effects, permitting more concise specifications. Intuitively, the unit I is an "empty" effect, which need not be a bottom element. $\top$ is an error (invalid effect, allowing us to reason about "undefined" effect sequences).

Effect quantales inherit a rich equational theory of semilattices, monoids, and extensive study of ordered algebraic systems [6, 23, 7, 22] from their several substructures, providing many ready-to-use properties for simplifying complex effects, and giving rise to other properties more interesting to our needs. One such example is an important and expected form of monotonicity property: that sequential composition respects the partial order on effects. In lattice-ordered monoids, this property is called isotonicity, and its proof for complete lattices [6, ch. 14.4] carries over directly to effect quantales because it requires only binary joins:

▶ **Proposition 2** (Isotonicity). In an effect quantale Q, $a \sqsubseteq b$ and $c \sqsubseteq d$ implies that $a \rhd c \sqsubseteq b \rhd d$.

▶ Proof. Because $b \rhd d = b \rhd (c \sqcup d) = (b \rhd c) \sqcup (b \rhd d)$, we know $b \rhd c \sqsubseteq b \rhd d$ by the definition of $\sqsubseteq$. Repeating the reasoning: $b \rhd c = (a \sqcup b) \rhd c = (a \rhd c) \sqcup (b \rhd c)$, so $a \rhd c \sqsubseteq b \rhd c$. The partial order is transitive, thus $a \rhd c \sqsubseteq b \rhd d$ ◻

An important litmus test for a general model of sequential effects is that it should subsume commutative effects (modeled as a join semilattice). This not only implies consistency of effect quantales with traditional effect systems, but ensures implementation frameworks for sequential effects (based on effect quantales) would be adequate for implementing commutative systems as well.

▶ **Lemma 3** (Subsumption of Commutative Effects). *Every commutative effect system modeled as a bounded join semilattice yields an effect quantale, such that ordering of individual effects is irrelevant, by using join for the monoid operation.*

▶ Proof. Assume a bounded join semilattice $L = (E, \vee, \top, \bot)$ of effects. Define a new effect quantale Q as $(E, \vee, \vee, \top, \bot)$ (i.e., reuse the join for the monoid). Q satisfies the distributivity requirements of the effect quantale definition, and naturally has $\bot$ as the monoid unit. ◻

4 Modeling Prior Sequential Effect Systems with Effect Quantales

Many of the axioms of effect quantales are not particularly surprising given prior work on sequential effect systems; one of this paper's contributions is recognizing that these axioms are sufficiently general to capture many prior instances of sequential effect systems. We show two prominent examples here in detail, and cite further examples.

4.1 Locking with Effect Quantales

A common class of effect systems is those reasoning about synchronization — which locks are held at various points in the program. In most systems this is done using scoped synchronization constructs, for which a bounded join semilattice is adequate. Here, we give an effect quantale for flow-sensitive tracking of lock sets including recursive acquisition. The main idea is to use a multiset of locks (modeled by $\mathcal{M}(S) = S \to \mathbb{N}$, where the multiplicity of a lock is the number of claims a thread has to holding the lock — the number of times it has acquired said lock) for the locks held before and after each expression. We use $\varnothing$ to denote the empty multiset (where all multiplicities are 0). We use join on multisets to produce least upper bounds on multiplicities, union to perform addition of multiplicities, and set difference for zero-limited subtraction.

▶ **Definition 4** (Synchronization Effect Quantale $\mathcal{L}$)**.** An effect quantale $\mathcal{L}$ for lock-based synchronization with explicit mutex acquire and release primitives is given by:

- $E = \mathcal{M}(L) \times \mathcal{M}(L) \uplus \mathsf{Err}$ for a set L of possible locks.
- $(a, a') \sqcup (b, b') = (a \vee a', b \vee b')$ when both effects acquire and release the same set of locks the same number of times: $b/b' = a/a'$ and $b'/b = a'/a$. Otherwise, the join is Err.
- $(a, a') \rhd (b, b')$ is (c, c') for the least c and c' where $a \subseteq c$, $b \subseteq ((c/(a/a')) \cup (a'/a))$ (b's holdings are contained in c less lock releases from the first action, plus the lock acquisitions from the first action), and $c' = (((c/(a/a')) \cup (a'/a))/(b/b')) \cup (b'/b)$ when such a pair exists, and Err otherwise.
- $\top = \mathsf{Err}$
- $I = (\varnothing, \varnothing)$

Intuitively, the pair represents the sets of lock claims before and after some action, which models lock acquisition and release. $\sqcup$ intuitively requires each "alternative" to acquire/release the same locks, while the set of locks held for the duration may vary (and the result assumes enough locks are held on entry — enough times each — to validate either element). This can be intuitively justified by noticing that most effect systems for synchronization require, for example, that each branch of a conditional may access different memory locations, but reject cases where one branch changes the set of locks held while the other does not (otherwise the lock set tracked "after" the conditional will be inaccurate for one branch, regardless of other choices). Sequencing two lock actions, roughly, pushes the locks required by the second action to the precondition of the compound action (unless such locks were released by the first action, i.e. in a/a'), and pushes locks held after the first action through the second — roughly a form of bidirectional framing.

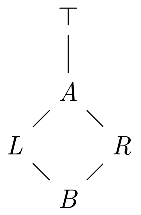
With this scheme, lock acquisition for some lock ℓ would have (at least) effect $(\varnothing, \{\ell\})$, indicating that it requires no locks to execute safely, and terminates holding lock ℓ. A release of ℓ would have swapped components — $(\{\ell\}, \varnothing)$ — indicating it requires a claim on ℓ to execute safely, and gives up that claim. Sequencing the acquisition and release would have effect $(\varnothing, \{\ell\}) \rhd (\{\ell\}, \varnothing) = (\varnothing, \varnothing)$. Sequencing acquisitions for two locks ℓ_1 and ℓ_2 would have effect $(\varnothing, \{\ell_1\}) \rhd (\varnothing, \{\ell_2\}) = (\varnothing, \{\ell_1, \ell_2\})$, propagating the extra claim on ℓ_1 that is not used by the acquisition of ℓ_2. This is true even when $\ell_1 = \ell_2 = \ell$ — the overall effect would represent the recursive acquisition as two outstanding claims to hold ℓ: $(\varnothing, \{\ell, \ell\})$.

A slightly more subtle example is the acquisition of a lock ℓ_2 just prior to releasing a lock ℓ_1, as would occur in the inner loop of hand-over-hand locking on a linked list: $(\mathsf{acquire}\ \ell_2;\ \mathsf{release}\ \ell_1)$ has effect $(\varnothing, \{\ell_2\}) \rhd (\{\ell_1\}, \varnothing) = (\{\ell_1\}, \{\ell_2\})$. The definition of $\rhd$ propagates the precondition for the release through the actions of the acquire; it essentially computes the minimal lock multiset required to execute both actions safely, and computes the final result of both actions' behavior on that multiset.

While use of sets rather than multisets would be simpler and would form an effect quantale for a given set of locks (with some use of disjoint union), such a formulation lacks an important property needed for substitution to behave correctly. We introduce that property in Section 6.1, and discuss subtle consequences of this in Section 9.3.

4.2 An Effect Quantale for Atomicity

One of the best-known sequential effect systems is Flanagan and Qadeer's extension of RCC/Java to reason about atomicity [20], based on Lipton's theory of *reduction* [40] (called *movers* in the paper). The details of the movers are beyond what space permits us to explain in detail, but the essential ideas were developed for a simpler language and effect system in an earlier paper [21], for which we give an effect quantale.



;	B	L	R	A	$\top$
B	B	L	R	A	$\top$
R	R	A	R	A	$\top$
L	L	L	$\top$	$\top$	$\top$
A	A	A	$\top$	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Figure 1 Atomicity effects [21]: lattice and sequential composition.

The core idea is that in a well-synchronized (i.e., data race free) execution, each action of one thread can be categorized by how it commutes with actions of other threads: a left (L) mover commutes left (earlier) with other threads' actions (e.g., a lock release), a right R mover commutes later (e.g., lock acquire), a both B mover commutes either direction (e.g., a well-synchronized field access). A sequence of right movers, then both-movers, then left-movers *reduces* to an atomic action (A). Repeating the process wrapping movers around an atomic action can again reduce to an atomic action, verifying atomicity for even non-trivial code blocks including multiple lock acquisitions. As a regular expression, any sequence of movers matching the regular expression $(R^*B^*)^*A(B^*L^*)^*$ reduces to an atomic action. Effect trace fragments of this form demarcate expressions that evaluate as if they were physically atomic.

▶ **Definition 5** (Atomicity Effect Quantale $\mathcal{A}$)**.** The effect quantale for Flanagan and Qadeer's simpler system [21] can be given as:

- $E = \{B, L, R, A, \top_{FQ}, \mathsf{Err}\}$. Note that $\top_{FQ}$ is the top effect in Flanagan and Qadeer's work — their use does not itself require an error element.
- $a \sqcup b$ is defined according to the lattice given by Flanagan and Qadeer [20] (Figure 1) augmented with the new Err element as top (not shown in Figure 1).
- $a \rhd b$ is defined according to Flanagan and Qadeer's ; operator (Figure 1) plus our added Err element as an annihilator ($\mathsf{Err} \rhd a = \mathsf{Err} = a \rhd \mathsf{Err}$).
- $\top = \mathsf{Err}$
- $I = B$

Flanagan and Qadeer also define an iterator operator on atomicities, used for ascribing effects to loops whose bodies have a particular atomicity. We defer iteration until Section 5, but will revisit this operator there.

Of course the atomicity effect quantale alone is insufficient to ensure atomicity, because atomicity depends on correct synchronization. The choice of effect for each program expression is not insignificant, but full atomicity checking requires the *product* of the synchronization and atomicity effect quantales. Thus, in Section 8 we study an *sequential extension* to Flanagan and Qadeer's work using $\mathcal{L} \otimes \mathcal{A}$.

▶ **Definition 6** (Products of Effect Quantales ($\otimes$))**.** The product $Q \otimes R$ of effect quantales Q and R is given by the product of the respective carrier lattices, with all pairs containing $\top$ from either constituent lattice merged into one single Err element for the new lattice. Other operations are lifted pointwise to each half of the product, modified so if the lifting of an original operation from Q or R produces $\top$ in the respective lattice, the operation in the product produces Err (in the product lattice). Identity is (I_Q, I_R), and top is Err.

4.3 Other Examples

Our running example of tracking recursive lock ownership and atomicity is one of the better-known sequential effect systems, but many more exist. We are unaware of a source-level sequential effect system that *does not* form an effect quantale.

A particularly important class of these examples are those that reason specifically about execution traces. All sequential effect systems reason to some degree about execution history, but some examples from the literature have very expressive notions of the past. Skalka's trace effects [52] are (abstractions of) sets of event traces, with trace concatenation (lifted to sets) as the monoid, and set union as join — these operations distribute as required by effect quantales, so adding a synthetic (unused) error element to Skalka's work produces one. Setting aside parallel composition (which we do not study), Nielson and Nielson's earlier communication effect system for Concurrent ML [46] is similar to Skalka's. Their *behaviors* act as trace set abstractions, with sequencing and non-deterministic choice (union) acting as an effect quantale's monoid and join operations. (They also include a separate parallel composition of behaviors we do not model, discussed in Section 7.3.) Their subtyping rules for behaviors imply the required distributivity laws (though as with Skalka's system, we must add a synthetic error element). Similarly, Koskinen and Terauchi [38] use pairs of trace sets characterizing the behavior of finite and infinite executions separately. Their effects form an effect quantale, though they additionally exploit set intersection for intersection effects.

5 Iteration

Many sequential effect systems include a notion of iteration, used for constructs like explicit loops. The operator for this, usually written as a postfix *, gives the overall effect of any (finite) number of repetitions of an effect.

The iteration construct must follow from some fixed point construction on the semilattice. However, the most obvious approach — using a least fixed point theorem on effect quantales with a bottom element — lacks an important property. Instead, we detail an approach based on *closure operators* on partially ordered sets in Section 5.2, which applies to any effect quantale satisfying some mild restrictions and coincides with manual iteration definitions for prior work. First, in Section 5.1, we motivate a number of required properties for any derived notion of iterating an effect.

5.1 Properties Required of an Iteration Operator

Iteration operators must satisfy a few simple but important properties to be useful. We first list, then explain these properties.

$P1 : \forall\, e.\, e \sqsubseteq e^*$ $\quad P2 : \forall\, e.\, e \rhd e^* \sqsubseteq e^*$ and $e^* \rhd e \sqsubseteq e^*$ $\quad P3 : \forall\, e.\, (e^*)^* = e^*$

$P4 : \forall\, e, f.\, (e \sqcup f)^* = e^* \sqcup f^*$ $\quad P5 : \forall\, e.\, I \sqsubseteq e^*$

Property P1 ensures one iteration of a loop body has no greater effect than multiple iterations. Similarly, the exact number of iterations should be immaterial (P2). Nesting should not matter, since semantically the nested loop structure is dynamically unrolled to some number of sequential iterations (P3). And P4 ensures certain equivalent ways of writing programs (e.g., a loop in each branch of a conditional vs. a conditional inside a loop) are effect-equivalent. P1 and P2 are essential validity requirements for iteration. P3 and P4 are not strictly necessary, but permit many additional effect simplifications and figure prominently in prior work that found them important for building manageable effect systems [21, 20]. P5 is slightly less obvious, but also critical: the least upper bound of the empty effect and some iterated effect should be the iterated effect. This allows some helpful simplifications on effects (e.g., for a conditional whose only non-trivial branch contains a loop), but will play an essential role in the soundness proof later (the effect of not executing a loop is I). This is also the property that fails for any straightforward use of least fixed point constructions — all such constructions work on ascending chains rooted at $\bot$ (therefore requiring a bottom element), but unless I is constrained to be $\bot$, there is no simple way to ensure with the fixed point equation alone that the resulting fixed point will be ordered above I. Such a constraint is not unheard of ($\mathcal{A}$ satisfies it), but not universal. $\mathcal{L}$ has no natural $\bot$, and adding a synthetic $\bot$ with identity behavior would mean introducing an effect usable for acquiring locks *or not acquiring locks*, which is undesirable.

5.2 Iteration via Closure Operators

For a general notion of iteration, we will use a *closure operator* on a poset:

▶ **Definition 7** (Closure Operator [6, 7, 51]). A closure operator on a poset $(P, \sqsubseteq)$ is a function $f : P \to P$ that is

- Extensive: $\forall\, e, e \sqsubseteq f(e)$
- Idempotent: $\forall\, e, f(f(e)) = f(e)$
- Monotone: $\forall\, e, e'.\, e \sqsubseteq e' \Rightarrow f(e) \sqsubseteq f(e')$

Closure operators have several particularly useful properties [6, 7, 51]:

- Idempotence implies that the range of a closure operator is also the set of fixed points of the operator.
- Closure operators on a poset are equivalent to their ranges. In particular, from the range of a poset, we can recover the original closure operator by mapping each element of the poset to the least element of the range that is above that input.
- A given subset of a poset is the range of a closure operator — called a *closure subset* — if and only if for every element x in the poset, every intersection with the principle up-set of x ($x\uparrow = \{y \mid x \sqsubseteq y\}$) has a bottom element [7, Theorem 1.8]. (The left direction of the iff is in fact proven by constructing the closure operator as described above.)

This means that if we can identify the desired range of our iteration operation (the results of the iteration operator) and show that it meets the criteria to be a closure subset, the construction above will yield an appropriate closure operator, which we can take directly as our iteration operation. For this to work, we must identify the desired range, and show it meets the requirements to induce a closure operator.

The natural choice is the set of elements for which sequential composition is idempotent, which we refer to as the *freely iterable elements*:

▶ **Definition 8** (Freely Iterable Elements). The set of freely iterable elements $\mathsf{Iter}(Q)$ of an effect quantale Q is defined as $\mathsf{Iter}(Q) = \{a \in Q \mid a \rhd a = a\}$.

To induce a closure operator for this set, we must show it exists, and that it is in fact a closure subset. The first is straightforward since Err and I are freely iterable:

▶ **Proposition 9** (Non-emptiness of Freely Iterable Elements). For any effect quantale Q, $\mathsf{Iter}(Q)$ is non-empty.

In general, the freely iterable elements do not themselves form a closure subset. They could fail to form a closure subset in the case where some element x is less than two incomparable freely iterable elements y and z, but x is not itself freely iterable and there is no other freely iterable element between — there is no freely iterable q such that $x \sqsubseteq q$ and $q \sqsubseteq y$ and $q \sqsubseteq z$. Phrased differently the intersection of some element's principle up-set and the freely iterable elements lacks a least element.

To derive our final solution, two further restrictions are required. First, the elements of our chosen closure subset must all reside at or above the identity in the semilattice, to ensure iteration permits loops to not execute. Second, $\mathsf{Iter}(Q)$ must be closed under joins:

$\forall\, x, y \in \mathsf{Iter}(Q).\, (x \sqcup y) \in \mathsf{Iter}(Q)$. This ensures iteration distributes over joins. We call such effect quantales — which have well-behaved closure operators — *iterable effect quantales.*

▶ **Definition 10** (Iterable Effect Quantale)**.** An effect quantale Q is *iterable* if and only if for all x the set $x\uparrow \cap (I \uparrow \cap \mathsf{Iter}(Q))$ contains a least element and $\mathsf{Iter}(Q)$ is closed under joins.

Another way to read the first part of the definition is that the closure operator will only exist for effect quantales for where, for every element x, if x is $\sqsubseteq$ two incomparable freely iterable elements y and z (each greater than I), then there is some freely iterable element $q \sqsupset I$ such that $x \sqsubseteq q \sqsubseteq y$ and $q \sqsubseteq z$ (possibly x itself).

Not all effect quantales are iterable, since the subset of freely iterable elements may not be closed under joins, and the semilattice may not contain a unique least freely iterable element greater than I for each possible effect. However, violating these appears uncommon; we have not observed it for any effect quantales we constructed, and cannot identify any systems in the literature with such irregular lattices. So in practice these restrictions on the existence of a closure-operator-based iteration appears unproblematic.

▶ **Proposition 11** (Closure for Iterable Effect Quantales)**.** For any iterable effect quantale Q, $I \uparrow \cap \mathsf{Iter}(Q)$ is a closure subset.

▶ Proof. $I \uparrow \cap \mathsf{Iter}(Q)$ is always non-empty, because $\mathsf{Iter}(Q)$ is non-empty and contains Err (Proposition 9), and $I \sqsubseteq \mathsf{Err}$. So if for every x, $x \uparrow \cap (I \uparrow \cap \mathsf{Iter}(Q))$ has a least element, $I \uparrow \cap \mathsf{Iter}(Q)$ is a closure subset [7, Theorem 1.8]. This requirement is exactly the meaning of Q being iterable, so this is a closure subset. □

▶ **Proposition 12** (Free Closure Operator on Iterable Effect Quantales)**.** For every iterable effect quantale Q, the function $F(X) \mapsto \mathsf{min}(X \uparrow \cap (I \uparrow \cap \mathsf{Iter}(Q)))$ is a closure operator satisfying properties P1–P5.

Our technical report [26] gives the proof, but note P1–3 follow from closure operator properties, P4 follows from $\mathsf{Iter}(Q)$'s join-closure, and P5 follows from using only elements of $I\uparrow$.

5.3 Iterating Concrete Effects

We briefly compare the results of applying our derived iteration operation to effect quantales we have discussed to known iteration operations.

▶ **Example 13** (Iteration for Atomicity)**.** The atomicity quantale $\mathcal{A}$ is iterable, so the free closure operator models iteration in that quantale. The result is an operator that is the identity everywhere except for the atomic effect A, which is lifted to $\top_{FQ}$ when repeated (it is not an error, but no longer atomic). This is precisely the manual definition Flanagan and Qadeer gave for iteration. In Section 4.2, we claimed any trace fragment matching a regular expression evaluated as if it were physically atomic — a property proven by Flanagan and Qadeer. In terms of effect quantales, this is roughly equivalent to the claim that $(R^* \rhd B^*)^* \rhd A \rhd (B^* \rhd L^*)^* \sqsubseteq A$. With our induced iteration operator, this has a straightforward proof:

$$\begin{array}{lll}
(R^* \rhd B^*)^* \rhd A \rhd (B^* \rhd L^*)^* & = (R \rhd B)^* \rhd A \rhd (B \rhd L)^* & \text{since } R^* = R,\ B^* = B \\
& = R^* \rhd A \rhd L^* & B \text{ is unit for } \rhd \\
& = R \rhd A \rhd L & \text{since } R^* = R,\ B^* = B \\
& = A & \text{by definition of } \rhd
\end{array}$$

▶ **Example 14** (Iteration for Commutative Effect Quantales)**.** For any bounded join semilattice, we have by Lemma 3 a corresponding effect quantale that reuses join for sequencing (and thus,

$\bot$ for unit), making the sequencing operation commutative. For purposes of iteration, this immediately makes all instances of this free effect quantale iterable, as idempotency of join ($x \sqcup x = x$) makes all effects freely iterable. The resulting iteration operator is the identity function, which exactly models the standard type rule for imperative loops in commutative effect systems, which reuse the effect of the body as the effect of the loop:

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} : \chi_1 \qquad \Gamma \vdash e_2 : \mathsf{unit} \mid \chi_2}{\Gamma \vdash \mathsf{while}(e_1)\{e_2\} : \mathsf{unit} \mid \chi_1 \sqcup \chi_2}$$

For a quantales where sequencing is merely the join operation on the semilattice, the above standard rule can be derived from our rule in Section 6 by simplifying the result effect:

$$\chi_1 \rhd (\chi_2 \rhd \chi_2)^* = \chi_1 \rhd (\chi_2 \rhd \chi_2) = \chi_1 \sqcup (\chi_2 \sqcup \chi_2) = \chi_1 \sqcup \chi_2$$

▶ **Example 15** (Loop Invariant Locksets)**.** For the lockset effect quantale $\mathcal{L}$, the freely iterable elements are all actions that do not acquire or release any locks — those of the form (a, a) for some a, and $\top$. These are isomorphic to the set of all multisets formed over the set of locks (plus the error element $\top$), and for those elements the join is equivalent to the complete lattice under multiset inclusion (again, plus the top error element). Since I is $(\varnothing, \varnothing)$ (which has no elements below it), $I \uparrow \sqcap \mathsf{Iter}(\mathcal{L}) = I \uparrow \sqcap(\{(a, b) \mid a = b\} \cup \{\top\}) = \{(a, b) \mid a = b\} \cup \{\top\}$. Because the freely iterable elements above unit form a *complete* lattice, $\mathcal{L}$ is iterable. The resulting closure operator is the identity on the freely iterable elements, and takes all actions that acquire or release locks to $\top$ (Err). This is exactly what intuition suggests as correct — the iterable elements are those that hold the same locks before and after each loop iteration, and attempts to repeat other actions should be errors.

6 Syntactic Type Soundness for Generic Sequential Effects

In this section we give a purely syntactic proof that effect quantales are adequate for syntactic soundness proofs of sequential type-and-effect systems. For the growing family of algebraic characterizations of sequential effect systems, this is the first soundness proof we know of that is (1) purely syntactic, (2) handles the indexed versions of the algebra required for singleton effects, (3) addresses effect polymorphism, and (4) includes direct iteration constructs. This development both more closely mirrors common type soundness developments for applied effect systems than the category theoretic approaches discussed in Section 7, and demonstrates machinery which would need to be developed in an analogous way for semantic proofs using those concepts. Thus, for hypothetical future effect systems requiring more flexibility than effect quantales provide, our techniques provide guidance on those concepts without switching to category theoretic denotational semantics.

We give this soundness proof for an *abstract* effect system — primitive operations, the notion of state, and the overall effect systems are all abstracted by a set of parameters (operational semantics for primitives that are aware of the state choice). This alone requires relatively little mechanism at the type level, but we wish to not only demonstrate that effect quantales are sound, but also that they are adequate for non-trivial existing sequential effect systems. In order to support such embeddings (see Sections 4 and 8), the type system includes parametric polymorphism — over types and effects as different kinds — as well as singleton types (e.g., for reference types with region tags) and effect constructors (such as effects mentioning particular locks). We consider effects equal according to the equations induced by effect quantale properties, and for families of effects indexed by values we identify the families with uses of appropriate effect constructors applied to singleton types. We demonstrate embeddings by directly translating equivalent constructs, and building artificial terms to model other constructs. These artificial terms' *derived* type rules directly match the language we embed, though the dynamic semantics may not be preserved (for example, we do not model concurrency). While unsuitable for a general framework in the style of a language workbench, this is adequate to show that our characterization of sequential effect systems' structure is flexible.

The language we study includes no built-in means to introduce a non-trivial (non-identity) effect, relying instead on the supplied primitives. The language also includes only the simplest form of parametric polymorphism for effects (and types), without bounding, constraints [30], relative effect declarations [59, 50], qualifier-based effects [27], or any other richer forms of polymorphism. Our focus is demonstrating compatibility of effect quantales with effect polymorphism and singleton effects, rather than to build a particularly powerful framework.

We stage the presentation to first focus on core constructs related to effect quantales, then briefly recap machinery from Systems F and Fω (and small modifications beyond what is standard), before proving type soundness. Section 8 gives an embedding from Flanagan and Qadeer's sequential effect system for atomicity [21] into our core language to establish that it is not only sound, but expressive.

6.1 Parameters to the Language

We parameterize our core language by a number of external features. First among these, is a slight extension of an effect quantale — an *indexed* effect quantale.

▶ **Definition 16** (Indexed Effect Quantale). An indexed effect quantale is a quantale whose elements (and therefore operations) are parameterized over some set.[2]

The lock set effect quantale $\mathcal{L}$ we described earlier is in fact an indexed effect quantale, parameterized by the set of lock names to consider.

Because the set of well-typed values changes during program execution, we will need to transport terms well-typed under one use of the quantale into another use of the quantale, under certain conditions. The first is the introduction of new well-typed values (e.g., from allocating a new heap cell), requiring a form of inclusion between indexed quantales. The second is due to substitution: our language considers variables to be values, but during substitution some variable may be replaced by another value that was already present in the set. This essentially collapses what statically appears as two values into a single value, thus *shrinking* the set of values distinguished inside the quantale. Each requires a different kind of homomorphism between effect quantales, with different properties.

▶ **Definition 17** (Effect Quantale Homomorphism). An *effect quantale homomorphism* between two effect quantales Q and R is a join semilattice homomorphism (a function between the carrier sets that preserves joins) that additionally preserves sequencing and $\top$.

▶ **Definition 18** (Monotone Indexed Effect Quantale). An indexed effect quantale Q is called *monotone* when for two sets S and T where $S \subseteq T$, the inclusion function from the carrier of $Q(S)$ to the carrier set of $Q(T)$ induces the obvious inclusion homomorphism.

[2] For those accustomed to typed meta-logics (e.g., Coq), one could view an indexed quantale as roughly the type $\forall\,\alpha : \mathsf{Type}.\,\{\mathsf{Decidable}\ \alpha\} \rightarrow \alpha \rightarrow \mathsf{Quantale}$. The point is that the details of the set are irrelevant to the quantale's definition.

▶ **Definition 19** (Collapsible Indexed Effect Quantale). An indexed effect quantale Q is called *collapsible* when for any non-empty set S and additional element x (not in S), a function f from $S \cup \{x\}$ to S that is the identity on elements of S induces a corresponding homomorphism where only sequences and joins that produced $\top$ under $S \cup \{x\}$ produce $\top$ when transported by the homomorphism (i.e., $f(a) \rhd f(b) = \top \Rightarrow a \rhd b = \top$, similarly for joins).

We parameterize our core language by a monotone, collapsible indexed effect quantale Q. Monotonicity is a natural requirement, but collapsibility has some subtle consequences we defer to Section 9. Any constant (i.e., non-indexed) effect quantale trivially lifts to a monotone collapsible indexed effect quantale that ignores its arguments. The product construction $\otimes$ lifts in the expected way.

The language parameters also include:

- An abstract notion of state, usually noted by $\sigma \in \mathsf{State}$. For a pure calculus State might be unit, while other languages might instantiate it to a heap, etc.
- A set of primitives p_i operating on terms and States. This includes modeling additional values that do not interact directly with general terms, such as references.
- A set of type families T_i for describing the types of primitives.
- A meta-function K for ascribing appropriate kinds to types in T_i. Thus, reference types may be modeled this way.
- A meta-function δ for ascribing a type to some primitive that is independent of the state — i.e., source-level primitive operations (but not store references). δ is constrained such that for values whose types are applicative (i.e., function types and quantified types) only the very last such type may have non-unit effect.
- A partially ordered state type environment $\Sigma \in \mathsf{StateEnv}$, which maps a subset of the primitives to types. The least element in the partial order is δ (used for source typing of primitives).
- A *partial* primitive semantics $[\![-]\!] : \mathsf{Term} \rightharpoonup \mathsf{State} \rightarrow \mathsf{Term} \times E \times \mathsf{State}$ defined only on full applications of primitive operations (i.e., fully-applied primitive operations, judged according to the types from δ).

For type soundness, we will rely on the following:

- Types produced by δ must be well-formed in the empty environment, and must not be closed base types (e.g., the primitives cannot add a third boolean, which would break the canonical forms lemma).
- Effects produced by $[\![-]\!]$ are valid for the quantale parameterized by the values at the call site (i.e., the dynamic effects depend only on the values at the call).
- There is a relation $Q \vdash \sigma : \Sigma$ for well-typed states.
- When the primitive semantics are applied to well-typed primitive applications and a well-typed state, the resulting term is well-typed (in the empty environment) with argument substitutions applied, and the resulting state is well-typed under some "larger" state type:
$$\begin{aligned}&\epsilon;\ \Sigma \vdash p_i\ \overline{v} : \tau \mid \gamma \wedge Q \vdash \sigma : \Sigma \wedge [\![p_i\ \overline{v}]\!](\sigma) = (v', \gamma', \sigma')\\ &\qquad \Rightarrow \exists\, \Sigma'.\, \Sigma \leq \Sigma' \wedge \epsilon;\ \Sigma' \vdash v' : \tau[\overline{v}/\mathsf{args}(\delta(p_i))] \mid I \wedge Q \vdash \sigma' : \Sigma'\end{aligned}$$
We call this property *primitive preservation.*

This setup leads to a delicate dependency order among these parameters and the core language to avoid circularity. Such circularity is manageable with sophisticated tools in the ambient logic [12, 5], but we prefer to avoid them for now. The parameters and language components are stratified as follows:

- The syntax of kinds is closed.

$$
\begin{array}{llcl}
\textsf{Kinds} & \kappa & ::= & \star \mid \mathcal{E} \mid \kappa \Rightarrow \kappa \\
\textsf{Types} & \tau & ::= & T_i \mid \tau\ \tau \mid E_Q \mid \Pi x : \tau \xrightarrow{\tau} \tau \mid \alpha \mid \mathsf{bool} \mid \forall\, \alpha :: \kappa \xrightarrow{\tau} \tau \mid \mathsf{unit} \mid \mathcal{S}(v) \\
\textsf{Terms} & e & ::= & p_i \mid (\lambda\, x.\, e) \mid e\ e \mid x \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ e\ e \mid \mathsf{while}\ e\ e \mid (\Lambda\alpha :: \kappa.\, e) \mid e[\tau] \mid () \\
\textsf{TypeEnv} & \Gamma & ::= & \epsilon \mid \Gamma, x : \tau \mid \Gamma, \alpha :: \kappa \\
\textsf{Values} & v & ::= & p_i \mid (\lambda\, x.\, e) \mid x \mid \mathsf{true} \mid \mathsf{false}
\end{array}
$$

$$
\boxed{\vdash \Gamma} \qquad \frac{}{\vdash \epsilon} \qquad \frac{\Gamma \vdash \tau :: \star \qquad x \notin \Gamma}{\vdash \Gamma, x : \tau} \qquad \frac{\alpha \notin \Gamma}{\Gamma \vdash \alpha :: \kappa}
$$

$$
\boxed{\Gamma \vdash \tau :: \kappa} \qquad \frac{}{\Gamma \vdash T_i :: K(T_i)} \qquad \frac{\Gamma(\alpha) = \kappa}{\Gamma \vdash \alpha :: \kappa} \qquad \frac{\Gamma \vdash \tau :: \kappa \Rightarrow \kappa' \qquad \Gamma \vdash \tau' :: \kappa}{\Gamma \vdash \tau\ \tau' :: \kappa'} \qquad \frac{E \in Q(\Gamma)}{\Gamma \vdash E :: \mathcal{E}}
$$

$$
\frac{\begin{array}{c}\Gamma \vdash \tau :: \star \\ \Gamma, x : \tau \vdash \gamma :: \mathcal{E} \\ \Gamma, x : \tau \vdash \tau' :: \star\end{array}}{\Gamma \vdash (\Pi x : \tau \xrightarrow{\gamma} \tau') :: \star} \qquad \frac{}{\Gamma \vdash \mathsf{bool} :: \star} \qquad \frac{}{\Gamma \vdash \mathsf{unit} :: \star} \qquad \frac{\Gamma \vdash v : \tau \mid I}{\Gamma \vdash \mathcal{S}(v) :: \star} \qquad \frac{\begin{array}{c}\Gamma, \alpha :: \kappa \vdash \gamma :: \mathcal{E} \\ \Gamma, \alpha :: \kappa \vdash \tau :: \star\end{array}}{\Gamma \vdash \forall\, \alpha :: \kappa \xrightarrow{\gamma} \tau :: \star}
$$

$$
\boxed{\Gamma \vdash e : \tau \mid \gamma} \qquad \frac{}{\Gamma \vdash p_i : \delta(p_i) \mid I} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \mid I} \qquad \frac{\Gamma, x : \tau \vdash e : \tau' \mid \gamma_e \qquad \gamma_e \sqsubseteq \gamma}{\Gamma \vdash (\lambda\, x.\, e) : \Pi x : \tau \xrightarrow{\gamma} \tau' \mid I}
$$

$$
\frac{\Gamma \vdash e_1 : \Pi x : \tau \xrightarrow{\gamma} \tau' \mid \gamma_1 \qquad \Gamma \vdash e_2 : \tau \mid \gamma_2 \qquad x \notin \mathsf{FV}(\gamma, \tau') \vee \mathsf{Value}(e_2)}{\Gamma \vdash e_1\ e_2 : \tau'[e_2/x] \mid \gamma_1 \rhd \gamma_2 \rhd \gamma[e_2/x]} \qquad \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash b : \mathsf{bool} \mid I}
$$

$$
\frac{\Gamma \vdash c : \mathbb{B} \mid \gamma_c \qquad \Gamma \vdash e_1 : \tau \mid \gamma_1 \qquad \Gamma \vdash e_2 : \tau \mid \gamma_2}{\Gamma \vdash \mathsf{if}\ c\ e_1\ e_2 : \tau \mid \gamma_c \rhd (\gamma_1 \sqcup \gamma_2)} \qquad \frac{\Gamma \vdash c : \mathsf{bool} \mid \gamma_c \qquad \Gamma \vdash e : \tau \mid \gamma_b}{\Gamma \vdash \mathsf{while}\ c\ e : \mathsf{unit} \mid \gamma_c \rhd (\gamma_b \rhd \gamma_c)^*}
$$

$$
\frac{\Gamma, \alpha :: \kappa \vdash e : \tau \mid \gamma}{\Gamma \vdash (\Lambda\alpha :: \kappa.\, e) : \forall\, \alpha :: \kappa \xrightarrow{\gamma} \tau \mid I} \qquad \frac{\Gamma \vdash e : \forall\, \alpha :: \kappa \xrightarrow{\gamma} \tau \mid \gamma_e \qquad \Gamma \vdash \tau' :: \kappa}{\Gamma \vdash e[\tau'] : \tau[\tau'/\alpha] \mid \gamma_e \rhd \gamma[\tau'/\alpha]} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit} \mid I}
$$

$$
\boxed{\sigma, e \rightarrow^{\gamma}_{Q} \sigma', e'} \qquad \frac{}{\sigma, (\lambda\, x.\, e)\ v \rightarrow^{I}_{Q} \sigma, e[v/x]} \qquad \frac{}{\sigma, (\Lambda\alpha :: \kappa.\, e)[\tau] \rightarrow^{I}_{Q} e[\tau/\alpha]} \qquad \frac{[\![p_i\ \overline{v}]\!](\sigma) = (e', \gamma, \sigma')}{\sigma, p_i\ \overline{v} \rightarrow^{\gamma}_{Q} \sigma', e'}
$$

$$
\frac{}{\sigma, \mathsf{if}\ \mathsf{true}\ e_1\ e_2 \rightarrow^{I}_{Q} \sigma, e_1} \qquad \frac{}{\sigma, \mathsf{if}\ \mathsf{false}\ e_1\ e_2 \rightarrow^{I}_{Q} \sigma, e_2} \qquad \frac{}{\sigma, \mathsf{while}\ e\ e_b \rightarrow^{I}_{Q} \sigma, \mathsf{if}\ e\ (e_b;\ \mathsf{while}\ e\ e_b)\ ()}
$$

$$
\boxed{\sigma, e \xrightarrow{\gamma}{}^{*}_{Q} \sigma', e'} \qquad \frac{}{\sigma, e \xrightarrow{I}{}^{*}_{Q} \sigma, e} \qquad \frac{\sigma, e \xrightarrow{\gamma}{}^{*}_{Q} \sigma', e' \qquad \sigma', e' \rightarrow^{\gamma'}_{Q} \sigma'', e''}{\sigma, e \xrightarrow{\gamma \rhd \gamma'}{}^{*}_{Q} \sigma'', e''}
$$

Figure 2 A generic core language for sequential effects, omitting straightforward structural rules from the operational semantics. ; is standard sugar for sequencing with in a CBV lambda calculus.

- The core language's syntax for terms and types is mutually defined (the language contains explicit type application and singleton types), parameterized by T_i and p_i. The latter parameters are closed sets, so the mutual definition is confined to the core.
- The type judgment depends on (beyond terms, types, and kinds) δ, K, and StateEnv.
- State may depend on terms, types, and kinds.
- The dynamic semantics will depend on terms, types, kinds, State, and $[\![-]\!]$ (which cannot refer back to the main dynamic semantics).
- Primitive preservation depends on the typing relation and state typing.
- The type soundness proof will rely on all core typing relations, state typing (which may be defined in terms of source typing), and the primitive preservation property.

Ultimately this leads to a well-founded set of dependencies for the soundness proof.

6.2 The Core Language, Formally

Figure 2 gives the (parameterized) syntax of kinds, types, and terms. Most of the structure should be familiar from standard effect systems and Systems F and Fω (with multiple kinds, as in the original polymorphic effect calculus [41]), plus standard while loops and conditionals with effects sequenced as in Section 2. We focus on the differences.

The language includes a dependent product (function) type, which permits program values to be used in types and effects. This is used primarily through effects — elements of an effect quantale may mention elements of the set — and through the singleton type constructor $\mathcal{S}(-)$, which associates a type (classifying no terms) with each program value. Use of the dependent function space is restricted to syntactic values (which includes variables in our call-by-value language) — the application rule requires that either the argument is a syntactic value, or the function type's named argument does not appear in the effect or result type. In the latter case, for concrete types we will use the standard $\tau \overset{\gamma}{\to} \tau'$ notation. A minor item of note is that dependent function types and quantified types bind their argument in the function's effect as well as in the result type. This permits uses such as a function acquiring the lock passed as an argument. One small matter important to the soundness proof: for any value, the effect of the value itself is the identity effect I.

Every rule carries an implicit side condition that the resulting effect is $\neq \top$. Since $\top$ acts as the error element, this permits effect systems to completely reject certain event orders.

A slightly more subtle point concerns the kinding judgment for effects. The requirement is that an effect E is valid if it is contained in $Q(\Gamma)$. This is because the type system is actually given with respect to an indexed effect quantale, as described above, which accepts some set to parameterize the system by. $Q(\Gamma)$ is Q instantiated with the set of well-typed values under Γ.

It is worth recalling briefly the role of parametric effect polymorphism and singleton types in our system. Singleton types are used as a way to specify elements of the effect quantale that depend on program values. They are used in type-level application with the effect constructors we assume are given for the effect quantale. They are also used for data types: for example, they are used to associate a reference type with the lock guarding access to that heap cell. Effect polymorphism is an essential aspect of code reuse in static effect systems [41, 55, 50, 27]. It permits writing functions whose effects depend on the effect of higher-order arguments. For example, consider the atomicity of fully applying the annotated term

$$\mathcal{T} = \lambda\,\ell : \mathsf{lock}.\,\Lambda\gamma :: \mathcal{E}.\,\lambda f : \mathsf{unit} \overset{\gamma}{\to} \mathsf{unit}.\,(\mathsf{acquire}\ \ell;\ f\ ();\ \mathsf{release}\ \ell)$$

The atomicity of a full application of term $\mathcal{T}$ (i.e., application to a choice of effect and appropriately typed function term) depends on the (latent) atomicity of f. For the moment, assume we track only atomicities (not lock ownership). The type of $\mathcal{T}$ is

$$\Pi\ell : \mathsf{lock} \overset{B}{\to} \forall\,\gamma :: \mathcal{E} \overset{B}{\to} (\mathsf{unit} \overset{\gamma}{\to} \mathsf{unit}) \xrightarrow{R \rhd \gamma \rhd L} \mathsf{unit}$$

If $f1$ performs only local computation, its latent effect can have static atomicity B, making the atomicity of $\mathcal{T}[B]\,f1$ atomic (A). If $f2$ acquires *and releases* locks, its static effect must be $\top_{FQ}$ (valid but non-atomic), making the atomicity of $\mathcal{T}[\top_{FQ}]\,f2$ also valid but non-atomic.

The operational semantics is mostly standard: a labeled transition system over pairs of states and terms, where the label is the effect of the basic step. We omit the structural rules that simply reduce a subexpression and propagate state changes and the effect label in the obvious way. The only other subtlety of the single-step relation is that when reducing invocations of primitives, if a primitive's semantics via $[\![-]\!]$ are defined only on larger-arity calls than what has been reduced to values $\overline{v}$ (which also includes type applications), the next argument applied is reduced, structurally. Incomplete applications of primitives remain stuck. We also give a transitive reduction relation $\xrightarrow{\gamma}{}^{*}_{Q}$ which accumulates the effects of each individual step.

Runtime Typing

Figure 2 gives the source type system. For the runtime type system, three changes are made. First, a state type Σ is added to the left side of each judgment in the standard way. Second, primitive typing is changed to rely on Σ rather than δ (recall that δ is the least element in the partial order, so all Σ will extend δ). And third, the effect kinding is modified to check for effects in $Q(\Gamma, \Sigma)$ — the effect quantale instantiated for a set of values well-typed under Γ and Σ, allowing values introduced at runtime (such as dynamically allocated locks or references) to appear in effects.

6.3 Syntactic Safety

Syntactic type safety proceeds in the normal manner (for a language with mutually-defined types and terms), with only a few wrinkles due to effect quantales. Here we give the statements of the major lemmas affected by effect quantales, and give relevant details. For more details and statements of other lemmas (canonical forms, substitution of types into types and terms, progress), see the technical report [26].

Substitution lemmas are proven by induction on the expression's type derivation, exploiting the fact that all values' effects before subeffecting are I:

▶ **Lemma 20** (Term Substitution). *If* $\Gamma, x : \tau \vdash e : \tau \mid \gamma$ *and* $\Gamma \vdash v : \tau \mid I$*, then* $\Gamma \vdash e[v/x] : \tau[v/x] \mid \gamma[v/x]$*, and simultaneously if* $\Gamma, x : \tau \vdash \tau' :: \kappa$ *and* $\Gamma \vdash v : \tau \mid I$ *then* $\Gamma \vdash \tau'[v/x] :: \kappa$*.*

▶ Proof. By simultaneous induction on the typing and kinding relations. The only subtle case is substitution of a variable occurring in an effect. In this case, the set of well-typed values is being reduced in size by one, with uses of the substituted variable being replaced by the new value. This induces the type of homomorphism relevant for collapsible (indexed) effect quantales. By assumption Q is collapsible, so applying the appropriate homomorphism as substitution yields an effect that is well-kinded in the smaller type environment. ◻

We give type preservation below, assuming an iterable effect quantale. This assumption is only used in while-related cases, so this proof also shows soundness for programs without loops under non-iterable quantales.

▶ **Lemma 21** (One Step Type Preservation). *For all* Q*,* σ*,* e*,* e'*,* Σ*,* τ*,* γ*, and* γ'*, if* $\epsilon;\ \Sigma \vdash e : \tau \mid \gamma$*,* $Q \vdash \sigma : \Sigma$*,* $\delta \leq \Sigma$*, and* $\sigma, e \to_{Q}^{\gamma'} \sigma', e'$ *then there exist* Σ'*,* γ'' *such that* $\epsilon;\ \Sigma' \vdash e' : \tau \mid \gamma''$*,* $Q \vdash \sigma' : \Sigma'$*,* $\Sigma \leq \Sigma'$*,* $\gamma' \rhd \gamma'' \sqsubseteq \gamma$*.*

▶ Proof. By induction on the reduction relation. We show here only the while loop case because it leans heavily on details of the iteration construct. See the technical report [26] for other cases.

- Case E-While: Here $e = \mathsf{while}\ e_c\ e_b$, $\gamma' = I$, $\sigma = \sigma'$, and $e' = \mathsf{if}\ e_c\ (e_b;\ (\mathsf{while}\ e_c\ e_b))\ ()$. By inversion on typing:

$$\epsilon;\ \Sigma \vdash e_c : \mathsf{bool} \mid \gamma_c \qquad \epsilon;\ \Sigma \vdash e_b : \tau_b \mid \gamma_b \qquad \gamma = \gamma_c \rhd (\gamma_b \rhd \gamma_c)^{*} \qquad \tau = \mathsf{unit}$$

By T-If, T-Unit, desugaring ; to function application, and weakening, $\epsilon;\ \Sigma \vdash \mathsf{if}\ e_c\ (e_b;\ (\mathsf{while}\ e_c\ e_b))\ ()\ : \mathsf{unit} \mid \gamma_c \rhd (((\gamma_b \rhd \gamma_c) \rhd (\gamma_b \rhd \gamma_c)^*) \sqcup I)$. State remains unchanged, so the final obligation in this case is to prove the effect just given for e' (technically, preceded by $I \rhd$) is a subeffect of $\gamma = \gamma_c \rhd (\gamma_b \rhd \gamma_c)^*$, which relies crucially on iteration properties P2 and P5:

$$\begin{array}{rcl} \gamma_c \rhd ((\gamma_b \rhd \gamma_c \rhd (\gamma_b \rhd \gamma_c)^*) \sqcup I) & \sqsubseteq & \gamma_c \rhd (((\gamma_b \rhd \gamma_c) \rhd (\gamma_b \rhd \gamma_c)^*) \sqcup I) \\ & \sqsubseteq & \gamma_c \rhd (((\gamma_b \rhd \gamma_c)^*) \sqcup I) \\ & \sqsubseteq & \gamma_c \rhd ((\gamma_b \rhd \gamma_c)^*) \end{array}$$

□

▶ **Theorem 22** (Type Preservation). *For all Q, σ, e, e', Σ, τ, γ, and γ', if $\epsilon;\ \Sigma \vdash e : \tau \mid \gamma$, $Q \vdash \sigma : \Sigma$, $\delta \leq \Sigma$, and $\sigma, e \overset{\gamma'}{\rightarrow}{}^{*}_{Q} \sigma', e'$, then there exist Σ', γ'' such that $\epsilon;\ \Sigma' \vdash e' : \tau \mid \gamma''$, $Q \vdash \sigma' : \Sigma'$, $\Sigma \leq \Sigma'$, and $\gamma' \rhd \gamma'' \sqsubseteq \gamma$.*

7 Relationships to Semantic Notions of Effects

Our notion of an effect quantale is motivated by generalizing directly from the form of effect-based type judgments. In parallel with our work, there has been a line of semantically-oriented work to generalize monadic semantics to capture sequential effect systems (indeed, this is where our use of the term “sequential effect system” originates). Here we compare to several recent developments: Tate’s productors (and algebraic presentation as effectoids) [56], Katsumata’s effect-indexed monads [36], and Mycroft, Orchard, and Petricek’s joinads (and algebraic presentation in terms of joinoids) [45].

All of this work is done primarily in the setting of category theory, by incrementally considering the categorical semantics of desirable effect combinations (in contrast to our work, working by generalizing actual effect systems). Fortunately, each piece of work also couples the semantic development with an algebraic structure that yields an appropriate categorical structure, and we can compare directly with those without appealing to much category theory. None of the following systems consider effect polymorphism or give more than a passing mention of iteration, though given the generality of the technical machinery, we cannot say any of the following are incompatible with these ideas — only that their use has not been considered. In contrast, we showed (Section 6) that effect quantales are compatible with these ideas. Effect domains that depend on program semantics (e.g., singleton effects) have also not been considered in this semantic work, while we consider indexed effect quantales whose effects depend on program values. Of the three families of semantic work we compare to, only Mycroft et al. go so far as to consider conditionals and discuss iteration, which are ignored (in favor of other important issues) in Tate and Katsumata’s work.

Overall, Tate and Katsumata’s work studies structures which are strict generalizations of effect quantales (i.e., impose fewer constraints than effect quantales), and any effect quantale can be translated directly to Tate’s effectoids or Katsumata’s partially ordered effect monoid. Tate and Katsumata demonstrate that their structures are *necessary* to capture certain parts of any sequential effect system — a powerful general claim. By contrast, we demonstrate that with just a bit more structure than either of these, effect quantales become *sufficient* to formalize a range of real sequential effect systems. Mycroft et al.’s work does consider a full programming language, but studies a different set of structures than we do (block-structured parallelism rather than iteration).

7.1 Productors and Effectoids

Tate [56] sought to design the maximally general semantic notion of sequential composition, proposing a structure called *productors*, and a corresponding algebraic structure for source-level effects called an *effector*. Effectors, however, include models of analyses that are not strictly modular (e.g., can special-case certain patterns in source code for more precise effects) [56, Section 5]. To model the strictly compositional cases like syntactic type-and-effect systems, he also defines a semi-strict variant called an *effectoid* (using slightly different notation):

▶ **Definition 23** (Effectoid [56]). An *effectoid* is a set $\textsc{Eff}$ with a unary relation $\mathsf{Base}(-)$, a binary relation $- \leq -$, and a ternary relation $- \fatsemi - \mapsto -$, satisfying

- Identity: $\forall \varepsilon, \varepsilon'. (\exists \varepsilon_\ell. \mathsf{Base}(\varepsilon_\ell) \land \varepsilon_\ell \fatsemi \varepsilon \mapsto \varepsilon') \Leftrightarrow \varepsilon \leq \varepsilon' \Leftrightarrow (\exists \varepsilon_r. \mathsf{Base}(\varepsilon_r) \land \varepsilon \fatsemi \varepsilon_r \mapsto \varepsilon')$
- Associativity: $\forall \varepsilon_1, \varepsilon_2 \varepsilon_3, \varepsilon. (\exists \overline{\varepsilon}. \varepsilon_1 \fatsemi \varepsilon_2 \mapsto \overline{\varepsilon} \land \overset{\varepsilon}{\fatsemi} \varepsilon_3 \mapsto \varepsilon) \Leftrightarrow (\exists \hat{\varepsilon}. \varepsilon_2 \fatsemi \varepsilon_3 \mapsto \hat{\varepsilon} \land \varepsilon_1 \fatsemi \hat{\varepsilon} \mapsto \varepsilon)$
- Reflexive Congruence:
  - $\forall \varepsilon. \varepsilon \leq \varepsilon$
  - $\forall \varepsilon, \varepsilon'. \mathsf{Base}(\varepsilon) \land \varepsilon \leq \varepsilon' \implies \mathsf{Base}(\varepsilon')$
  - $\forall \varepsilon_1, \varepsilon_2, \varepsilon, \varepsilon'. \varepsilon_1 \fatsemi \varepsilon_2 \mapsto \varepsilon \land \varepsilon \leq \varepsilon' \implies \varepsilon_1 \fatsemi \varepsilon_2 \mapsto \varepsilon'$

Intuitively, Base identifies effects that are valid for programs with "no" effect — e.g., pure programs, empty programs. Tate refers to such effects as *centric*. The binary relation $\leq$ is clearly a partial order for subeffecting, and $- \fatsemi - \mapsto -$ is (relational) sequential composition. The required properties imply that the effectoid's sequential composition can be read as a non-deterministic function producing the minimal composed effect *or any supereffect thereof*, given that the sequential composition relation includes left and right units for any effect, and that Base and the last position of composition respect the partial order on effects.

Given Tate's aim at maximal generality (while retaining enough structure for interesting reasoning about sequential composition), it is perhaps unsurprising that all but the most degenerate effect quantale yields an effectoid by flattening the monoid and semilattice structure into the appropriate relations:

▶ **Lemma 24** (Quantale Effectoids). *For any nontrivial effect quantale Q (one with more elements than $\top$), there exists an effectoid E with the following structure:*

- $\textsc{Eff} = E/\{\top\}$
- $\mathsf{Base}(a) \overset{def}{=} I \sqsubseteq a \land a \neq \top$
- $a \leq b \overset{def}{=} a \sqsubseteq b \land b \neq \top$
- $a \fatsemi b \mapsto c \overset{def}{=} a \rhd b = c' \land c' \sqsubseteq c \land c \neq \top$

▶ Proof. The laws follow almost directly from the effect quantale laws. In the identity property, both left and right units are always chosen to be I. Associativity follows directly from associativity of $\rhd$ and isotonicity. The reflexive congruence laws follow directly from the definition (and transitivity) of $\sqsubseteq$. Note that we removed the top (error) element, representing failure by missing entries in the relations. □

A bit more surprising, perhaps, is that many effectoids directly yield quantales:

▶ **Lemma 25.** *For any effectoid E with a least centric element, and whose underlying partial order is a join semilattice, and which has a least result for any defined sequential composition, there exists an effect quantale Q such that:*

- $E_Q = \textsc{Eff}_E \uplus \mathsf{Err}$
- $\top = \mathsf{Err}$ *(a synthetic error element)*
- $\sqcup$ *performs the assumed binary join extended for new top element* Err*.*

- *$a \rhd b$ produces the least c such that $a \fatsemi b \mapsto c$ when defined, or* Err *when there is no such c such that $a \fatsemi b \mapsto c$ (by assumption, $a \fatsemi b$ is undefined or has a least element).*
- *I is assumed the least centric element*

Tate calls effectoids with a least result for any defined sequential composition *principalled*, and notes that they are common.

Essentially, in the case where the effectoid's partial order corresponds to a join semilattice with a single unit for sequencing and deterministic (modulo subsumption) sequencing, the two notions coincide. This strongly suggests that our generalization from the type judgments of a few specific effect systems, rather than from semantic notions, did not cost much in the way of generality. It also clarifies exactly when effectoids are more general: when effects form a partial order but *not* a join semilattice (no unique least upper bound of any pair), have no universal unit for sequencing, or have non-deterministic sequencing results. We are unaware of any complete source-level type-and-effect system with these properties.

7.2 Effect-indexed Monads, a.k.a. Graded Monads

Katsumata [36] pursues an independent notion of general sequential composition, where effects are formalized semantically as a form of type refining monad: a $T\ e\ \sigma$ is a monadic computation producing an element of type σ, whose effect is bounded by e (which classifies a subset of such computations). Based on general observations, Katsumata speculates that sequential effects form at least a pre-ordered monoid, and goes on to validate this (among other interesting results related to the notion of effects as refinements of computations). Katsumata shows categorically that these *effect indexed monads* (which later came to be known as *graded monads* to avoid confusion with other forms of indexing) are also a specialization of Tate's productors, exactly when the productor is induced by an effectoid derived from a partially-ordered monoid. Our notion of effect quantales directly induces a partially ordered monoid $(E, \sqcup, \rhd, I)$ satisfying the appropriate laws. However, the effectoid equivalent to this translation is not quite the same as the direct effectoid described earlier: graded monads (particularly the po-monoids) do not directly model partiality, while effectoids can. Setting this minor discrepancy aside (e.g., one could impose type system restrictions on its use, as we did in our type system) the relaxation between effect quantales and graded monads is due to relaxing the bounded join semilattice to a partial order, and the change from graded monads to effectoids (and thus productors) is due primarily to relaxing the rules for sequencing identity and determinism of sequencing. Katsumata does note briefly that many interesting effect systems rely on join-semilattices, but does not explore this specific class of graded monads in depth.

7.3 Joinads and Joinoids

Mycroft, Orchard, and Petricek [45] further extend graded monads to *graded conditional joinads*, and similar to Tate, give a class of algebraic structures — joinoids — that give rise to their semantic structures. As their base, they take graded monads, further assume a ternary conditional operator $? : (-,-,-)$ modeling conditionals whose branch approximation may depend on the conditional expression's effect, and parallel composition & suitable for fork-join style concurrency.

Their ternary operator is motivated by considerations of sophisticated effects such as control effects like backtracking (e.g., continuations). From their ternary operator, they derive a binary join, and therefore a partial order. However, their required laws for the ternary operator include only a right distributivity law because effects from the conditional expression itself do not in general distribute into the branches. Thus their derived semilattice structure satisfies only the right distributivity law $(a \sqcup b) \rhd c = (a \rhd c) \sqcup (b \rhd c)$, and not, in general, the left-sided equivalent. They also do not require "commutativity" of the branch arguments. This means that joinoids, in general, do not give rise to effect quantales — some (small) amount of structure is not necessarily present — and that in general they validate fewer equivalences between effects. An effect quantale can induce a ternary operator that ignores its first argument by simply taking the join of its other arguments, in which case Mycroft et al.'s derived partial order coincides with that derived from the quantale's join. As with the relation to graded monads this translates error element concretely rather than directly modeling partiality.

Joinads originally arose as an extension to monads that captures a class of combinators typical of composing parallel and concurrent programs in Haskell, in particular a *join* (unrelated to lattices) operator of type $M\ A \rightarrow M\ B \rightarrow M\ (A \times B)$. This is a natural model of fork-join-style parallel execution, and gives rise to the & operator of joinoids, which appears appropriate to model the corresponding notion in systems like Nielson and Nielson's effect system for CML communication behaviors [46], which is beyond the space of operations considered for effect quantales. However, & is inadequate for modeling the unstructured parallelism (i.e., explicit thread creation and termination, or task-based parallelism) found in most concurrent programming languages, so we did not consider such composition when deriving effect quantales. We would like to eventually extend effect quantales for unstructured concurrent programming: this is likely to include adapting ideas from concurrent program logics that join asynchronously [14], but any adequate solution should be able to induce an operation satisfying the requirements of joinoids' parallel composition.

Ultimately, any effect quantale gives rise to a joinoid, by using the effect quantale's join for both parallel composition and to induce the ternary operator outlined above.

Fixed Points

Mycroft et al. also give brief consideration to providing iteration operators through the existence of fixed points, noting the possibility of adding one type of fixed point categorically, which carried the undesirable side effect of requiring sequential composition to be idempotent: $\forall\, b.\, b \rhd b = b$. This is clearly too strict, and prohibits equivalents of both the lockset and atomicity effect quantales we studied. They take this as an indication that every operation should be explicitly provided by an algebra, rather than attempting to derive operators. By contrast, our closure operator approach not only imposes semantics that are by construction compatible with a given sequential composition operator, but critically coincide with manual definitions for existing systems.

7.4 Limitations of Semantics-Based Work

The semantic work on general models of sequential effect systems has not seriously addressed iteration. As discussed above, Mycroft et al. note that a general fixed point map could be added, but this forces $a \rhd a = a$ for all effects a, which is too restrictive to model the examples we have considered. Our approach to inducing an iteration operation through closure operators on posets should be generalizable to each of the semantic approaches we discussed. The semantics of such an approach are, broadly, well-understood, as closure operators on a poset are equivalent to a certain monad on a poset category; note that the three properties of closure operators — extensiveness, idemotence, and monotonicity — correspond directly to the formulation of a monad in terms of return, join (a flattening

$$\boxed{\Gamma \vdash e : a} \qquad \begin{array}{c}\text{EXP CONST}\\ \dfrac{}{\Gamma \vdash c : B}\end{array} \qquad \begin{array}{c}\text{EXP LOC}\\ \dfrac{}{\Gamma \vdash m : B}\end{array} \qquad \begin{array}{c}\text{EXP FUN}\\ \dfrac{\Gamma \vdash e : \Gamma(f)}{\Gamma \vdash f(\overline{x})e : B}\end{array} \qquad \begin{array}{c}\text{EXP PRIM}\\ \dfrac{\Gamma \vdash e_i : a_i}{\Gamma \vdash p(\overline{e}) : (a_1;\ \ldots;\ a_n;\ \Gamma(p))}\end{array}$$

$$\begin{array}{c}\text{EXP READ}\\ \dfrac{}{\Gamma \vdash x_\epsilon : B}\end{array} \qquad \begin{array}{c}\text{EXP RRACE}\\ \dfrac{}{\Gamma \vdash x_\bullet : A}\end{array} \qquad \begin{array}{c}\text{EXP ASSIGN}\\ \dfrac{\Gamma \vdash e : a}{\Gamma \vdash x_\epsilon := e : (a;\ B)}\end{array} \qquad \begin{array}{c}\text{EXP RASSIGN}\\ \dfrac{\Gamma \vdash e : a}{\Gamma \vdash x_\bullet := e : (a;\ A)}\end{array}$$

$$\begin{array}{c}\text{EXP LET}\\ \dfrac{\Gamma \vdash e_1 : a_1 \qquad \Gamma \vdash e_2 : a_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : (a_1;\ a_2)}\end{array} \qquad \begin{array}{c}\text{EXP IF}\\ \dfrac{\Gamma \vdash e : a \qquad \Gamma \vdash e_i : b : i}{\Gamma \vdash \mathsf{if}\ e\ e_1\ e_2 : (a;\ (b_1 \sqcup b_2))}\end{array} \qquad \begin{array}{c}\text{EXP WHILE}\\ \dfrac{\Gamma \vdash e_1 : a_1 \qquad \Gamma \vdash e_2 : a_2}{\Gamma \vdash \mathsf{while}\ e_1\ e_2 : (a_1;\ (a_2;\ a_1)^*)}\end{array}$$

$$\begin{array}{c}\text{EXP INVOKE}\\ \dfrac{\Gamma \vdash e : a \qquad \Gamma \vdash e_i : a_i}{\Gamma \vdash e^F(\overline{e}) : (a_1;\ \ldots;\ a_n;\ (\sqcup_{f \in F}\Gamma(f)))}\end{array} \qquad \begin{array}{c}\text{EXP FORK}\\ \dfrac{\Gamma \vdash e : a}{\Gamma \vdash \mathsf{fork}\ e : A}\end{array} \qquad \begin{array}{c}\text{EXP ATOMIC}\\ \dfrac{\Gamma \vdash e : a \qquad a \sqsubseteq A}{\Gamma \vdash \mathsf{atomic}\ e : a}\end{array}$$

Figure 3 Flanagan and Qadeer's type and effect system for atomicity of CAT programs.

operation $M(M\ A) \to M\ A$ unrelated to lattices or joinoids), and fmap. The semantic work discussed also omits treatment of polymorphism, and singleton or dependent types. As a result, their claim of adequacy for sequential effect systems is limited, whereas we have provide in Section 8 a direct implementation of a non-trivial composite sequential effect system in terms of effect quantales. On the other hand, their claims to generality are much stronger than ours, not only because the corresponding algebraic structures are less restrictive, but because they derived these structures by focusing on a few key elements common to all sequential effect systems (aside from the parallel combination studied for joinads) rather than directly attempting to generalize from concrete examples of sequential effect systems. Ultimately we view our work as strictly complementary to this categorical work — the latter is foundational and deeply general, while ours is driven by practice of sequential effect systems. Our work fills in a missing connection between these approaches and the concrete syntactic sequential effect systems most have studied.

The categorical semantics of polymorphism and dependent types (including singleton indexing as we have) are generally well-understood [47, 15, 35] and have even gained significant new tools of late [5], so the work discussed here should be compatible with those ideas, even if it requires adjustment. However, these related approaches would also need to be extended to account for substitution into effects that may mention program values; the notion of collapsibility will require an analogue in semantic accounts.

8 Modeling Prior Effect Systems in a Generic Framework

This section demonstrates that we can model significant prior type systems by embedding into our core language. Embedding here means a type-and-effect-preserving, but not necessarily semantics-preserving translation. Our language is generic, but clearly lacks concurrency, exception handling, and other concrete computational effects. Instead, we show how to model relevant primitives in our core language, giving derived type rules for those constructs, and translate type judgments to prove we would at least accept the same programs.

8.1 Types for Safe Locking and Atomicity

Here we briefly recall the details of Flanagan and Qadeer's earlier work on a type system for atomicity [21] (the full version [20] requires substantially more space and extends Java — modeling objects would require a more sophisticated type system for embedding). Flanagan and Qadeer's CAT language (Figure 3) is minimalist, defined in terms of a family of primitives (like our core language), with named functions, racing and race-free heap accesses, expected control constructs, and atomic blocks (which must be atomic). They use semicolons for

$$
\begin{array}{l}
Q(X) = \mathcal{L}(X) \otimes \mathcal{A} \\
M \in \mathsf{LockNames} \rightharpoonup \mathsf{Bool} \\
H \in \mathsf{Location} \rightharpoonup \mathsf{Term} \\
\mathsf{State} = M \times H \\
K(\mathsf{lock}) = \star \\
K(\mathsf{ref}) = \star \Rightarrow \star \Rightarrow \star
\end{array}
$$

$$
\frac{\forall\, l \in \mathsf{dom}(m).\, \Sigma(l) = \mathsf{lock} \qquad \forall\, r \in \mathsf{dom}(h).\, \epsilon;\ \Sigma \vdash h(r) : \Sigma(r) \mid I}{Q \vdash (m,h) : \Sigma}
$$

$$
\begin{array}{rcl}
\delta(\mathsf{new\_lock}) & = & \mathsf{unit} \xrightarrow{B} \mathsf{lock} \\
\delta(\mathsf{acquire}) & = & \Pi x : \mathsf{lock} \xrightarrow{(\varnothing,\{x\})\otimes R} \mathsf{unit} \\
\delta(\mathsf{release}) & = & \Pi x : \mathsf{lock} \xrightarrow{(\{x\},\varnothing)\otimes L} \mathsf{unit} \\
\delta(\mathsf{alloc}) & = & \Pi x : \mathsf{lock} \xrightarrow{B} \forall\, \alpha :: \star \xrightarrow{B} \tau \xrightarrow{B} \mathsf{ref}\ S(x)\ \tau \\
\delta(\mathsf{read}_\bullet) & = & \Pi x : \mathsf{lock} \xrightarrow{B} \forall\, \alpha :: \star \xrightarrow{B} \mathsf{ref}\ S(x)\ \tau \xrightarrow{(\varnothing,\varnothing)\otimes A} \tau \\
\delta(\mathsf{read}_\epsilon) & = & \Pi x : \mathsf{lock} \xrightarrow{B} \forall\, \alpha :: \star \xrightarrow{B} \mathsf{ref}\ S(x)\ \tau \xrightarrow{(\{x\},\{x\})\otimes B} \tau \\
\delta(\mathsf{write}_\bullet) & = & \Pi x : \mathsf{lock} \xrightarrow{B} \forall\, \alpha :: \star \xrightarrow{B} \mathsf{ref}\ S(x)\ \tau \xrightarrow{B} \tau \xrightarrow{(\varnothing,\varnothing)\otimes A} \tau \\
\delta(\mathsf{write}_\epsilon) & = & \Pi x : \mathsf{lock} \xrightarrow{B} \forall\, \alpha :: \star \xrightarrow{B} \mathsf{ref}\ S(x)\ \tau \xrightarrow{B} \tau \xrightarrow{(\{x\},\{x\})\otimes B} \tau \\
\delta(\mathsf{req\_atomic}) & = & (\mathsf{bool} \xrightarrow{A} \mathsf{unit}) \xrightarrow{B} \mathsf{unit}
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{new\_lock}\ \_]\!]((m,h))(\Sigma) = l, (m[l \mapsto \mathsf{false}], h), \Sigma[l \mapsto \mathsf{lock}] \text{ for next } l \notin \mathsf{dom}(m) \\
[\![\mathsf{acquire}\ l]\!]((m[l \mapsto \mathsf{false}], h))(\Sigma) = (), (m[l \mapsto \mathsf{true}], h), \Sigma \\
[\![\mathsf{release}]\!]((m[l \mapsto \mathsf{true}], h))(\Sigma) = (), (m[l \mapsto \mathsf{false}], h), \Sigma \\
[\![\mathsf{alloc}\ l\ \tau\ v]\!]((m,h))(\Sigma) = \ell, (m, h[\ell \mapsto v), \Sigma[\ell \mapsto \mathsf{ref}\ S(l)\ \tau] \text{ for } \ell \notin \mathsf{dom}(h) \\
[\![\mathsf{read}_\bullet\ \mathsf{l}\ \tau\ \ell]\!]((m,h))(\Sigma) = h(\ell), (m,h), \Sigma \\
[\![\mathsf{read}_\epsilon\ \mathsf{l}\ \tau\ \ell]\!]((m,h))(\Sigma) = h(\ell), (m,h), \Sigma \\
[\![\mathsf{write}_\bullet\ \mathsf{l}\ \tau\ \ell\ \mathsf{v}]\!]((m,h))(\Sigma) = v, (m, h[\ell \mapsto v]), \Sigma \\
[\![\mathsf{write}_\epsilon\ \mathsf{l}\ \tau\ \ell\ \mathsf{v}]\!]((m,h))(\Sigma) = v, (m, h[\ell \mapsto v]), \Sigma \\
[\![\mathsf{req\_atomic}\ f]\!]((m,h))(\Sigma) = (), (m,h), \Sigma
\end{array}
$$

Figure 4 Parameters to model Flanagan and Abadi's *Types for Safe Locking* [18] (a sequential variant) and Flanagan and Qadeer's *Types for Atomicity* [21] in our framework. We sometimes omit the locking component of effects when it is simply $(\varnothing, \varnothing)$ to improve readability.

sequencing of atomicity effects. For maximal minimalism, they assume some *other* type system has already analyzed the program and identified which heap accesses are racy and which are well-synchronized. For completeness, we will embed into an instantiation of our framework that itself distinguishes well-synchronized and racy reads, and establish conditions under which their abstract notion of well-synchronized is compatible. Thus this section develops a hybrid of Flanagan and Abadi's *Types for Safe Locking* [18] and Flanagan and Qadeer's *Types for Atomicity* [21], further extended to track locks in a flow-sensitive manner (the former uses `synchronized` blocks, the latter does not track locks itself). Recall that in the former, a concurrent functional language with heap is extended by locks, and the reference type is indexed by a singleton lock identity. The type system tracks the set of locks held at each program point (there, scoped by lexically scoped synchronized blocks), and ensures that any access to a heap location guarded by some lock occurs while that lock is held. This forms the foundation of the ideas behind the better-known RCC/Java [19], which extends these ideas to the full Java language. We add additional read and write primitives that may race, to model the atomicity work.

We define in Figure 4 the parameters to the language framework needed to model locks, mutable heap locations, and lock-indexed reference types, and the primitives to manipulate them. We define T_i by giving K (which is defined over T_i), and define p_i as $\mathsf{LockNames} \uplus \mathsf{Location} \uplus \mathsf{dom}(\delta)$ (locks, heap locations, and primitive operations). The state consists of a lock heap, mapping locks to a boolean indicating whether each lock is held, and a standard mutable store. The reference type is indexed by a lock (lifted to a singleton type). Primitives include lock allocation; lock acquisition and release primitives whose effects indicate both the change in lock claims and the mover type; allocation of data guarded by a particular lock; racing ($\bullet$) and well-synchronized (ϵ) reads and writes, with effects requiring (or not) lock ownership as appropriate; and one further primitive for requiring atomicity. The primitive types are largely similar, so we explain only two in detail. acquire takes one argument — a lock — that is then bound in the latent effect of the type. That effect is a product of the locking and atomicity quantales, indicating that the lock acquisition is a *right* *mover* (R), and that safe execution requires no particular lock claims on entry, but finishes with the guarantee that the lock passed as an argument is held (we use syntactic sugar for assumed effect constructors of appropriate arity). The read_ϵ primitive for well-synchronized (non-data-race) reads is akin to a standard dereference operator, but because it works for any reference — which may be associated with any lock and store values of any type — the choice of lock and type must be passed as arguments before the reference itself. Given the lock, cell type, and reference, the final latent effect indicates that the operation requires the specified lock to be held at invocation, preserves ownership, and is a *both mover* (B).

We give a stylized definition of the (partial) semantics function for primitives as acting on not only states but also state types, giving the monotonically increasing state type for each primitive, as required of the parameters. We also omit restating the dynamic effect in our $[\![-]\!]$; we take it to be the final effect of the corresponding entry in δ with appropriate value substitutions made — as required by the type system. The definitions easily satisfy the primitive preservation property assumed by the type system. We take as the partial order on $\mathsf{StateEnv}$ the standard partial order on partial functions, with δ as its least element.

These parameters are adequate to write and type terms like the following atomic function that reads from a supplied lock-protected reference (permitting syntactic sugar for brevity):

$$\begin{aligned}&\varnothing \vdash \lambda\, x.\, \lambda\, r.\, \mathsf{acquire}\ x;\ \mathsf{let}\ y = \mathsf{read}_\epsilon\ x\ [\mathsf{bool}]\ r\ \mathsf{in}\ (\mathsf{release}\ x;\ y)\\ &\quad : \left(\Pi x : \mathsf{lock} \xrightarrow{(\varnothing,\varnothing)\otimes B} \Pi r : \mathsf{ref}\ \mathcal{S}(x)\ \mathsf{bool} \xrightarrow{(\varnothing,\varnothing)\otimes A} \mathsf{bool}\right) \mid (\varnothing,\varnothing)\otimes B\end{aligned}$$

CAT is a properly multi-threaded language, while our language is not. As we noted earlier, our aim is to preserve well-typing, not dynamic semantics, so our translation of fork will not model concurrent semantics. Blocks of code that do not fork or rely on other threads should run as expected, though we do not prove this.

CAT's constants, primitives (`new_lock`, etc.), and mutexes can be translated in almost the obvious way for our framework, currying their primitives and extending that set with constants and the mutex names described above. The tricky bit is that CAT presumes some unspecified race freedom analysis and unspecified type system have already been applied to distinguish racing and well-synchronized reads, and to rule out basic type errors. Our terms require lock and type information to be explicitly present in the term, so we assume, beyond those unspecified analyses, operations $\mathsf{LockFor}$, $\mathsf{RefTypeOf}$, and TypeOf to extract the relevant local lock names and types. For a term produced using these operations to type-check in our core language will naturally require a degree of consistency between the unspecified analyses and the checks of our core language for the lock multiset quantale. However the details are not necessary to work out, because our relation is conditioned on the assumption that the translation does type check in our core language.

Conditionals and while loops are translated in the obvious inductive way — note that aside from CAT's type system lacking basic types, the handling of atomicity effects is structured exactly as our rules for those constructs. To handle currying, we adopt the notations $\lambda\, \overline{x}.\, e \equiv \lambda\, x_1 \ldots \lambda\, x_n.\, e$ for an n-ary closure, and $e\ \overline{e'} \equiv (\ldots(e\ e'_1)\ldots e'_n)$ for n-ary function application. Note that when typing the expanded forms, the effects of all but the innermost expanded lambda expression can simply be I, making the overall effect of the expanded application the left-to-right sequenced effects of the function and each argument followed by the effect of the inner-most closure. We also use the shorthand $\mathsf{wraplock}\ e \equiv \mathsf{let}\ x = \mathsf{new\_lock}()\ \mathsf{in}\ (\mathsf{acquire}\ x;\ \ e;\ \ \mathsf{release}\ x;\ ())$. The atomicity of this expression is A if and only if e's atomicity is less than A. Other translations are as follows, omitting analogous primitive translations:

$$\begin{array}{ll}
[\![\ p(\overline{e})\ ]\!] = p\ \overline{[\![e]\!]} & [\![\ e^F(\overline{e})\ ]\!] = [\![e]\!]\ \overline{[\![e]\!]} \\
[\![\ f(\overline{x})e\ ]\!] = (\lambda\, \overline{x}.\, [\![e]\!]) & [\![\texttt{atomic}\ e]\!] = \mathsf{req\_atomic}\ (\lambda\, \_.\, \mathsf{wraplock}\ [\![e]\!]);\ \ [\![e]\!] \\
[\![\texttt{fork}\ e]\!] = \mathsf{let}\ \_ = (\lambda\, \_.\, [\![e]\!])\ \mathsf{in}\ \mathsf{wraplock}\ () & [\![\ \ x_{\bullet}\ \ ]\!] = \mathsf{read}_{\bullet}\ \langle \mathsf{LockFor}(x)\rangle\ \langle \mathsf{RefTypeOf}(x)\rangle\ x
\end{array}$$

We assume the translation process produces a mapping from generated subterms back to the original CAT term (specifically, mapping closures back to CAT's named functions). `atomic` expression are translated to capture the expression in a dynamically-meaningless thunk passed as a parameter requiring an atomic effect, but run unconditionally. The unconditional execution allows the actual atomicity of e to be used later, as in CAT. `fork` operations are translated in a way that makes the forked thread computationally irrelevant (but, by induction, preserves typeability and effects) and locally carries an atomic effect as in the type rule.

The theorem we would like to prove is that translating any well-typed CAT term produces a term in our core language with the corresponding type and effect. Unfortunately, CAT is untyped aside from atomicities, so there is no type to translate, and CAT itself cannot check correct use of well-synchronized vs. racy reads. Instead, we prove an "un-embedding" lemma by induction on the CAT term:

▶ **Lemma 26** (Unembedding CAT from $\mathcal{L} \otimes \mathcal{A}$). *Given a* CAT *term t, for any Γ, τ, and effect $l \otimes e \in (\mathcal{L} \otimes \mathcal{A})(\Gamma)$ such that $\Gamma \vdash [\![t]\!] : \tau \mid l \otimes e$, under the* CAT *environment $\hat{\Gamma}$ mapping each function name to the final effect of its n-ary closure translation, $\hat{\Gamma} \vdash t : e$.*

9 Related and Future Work

The closely related work is split among three major groups: generic effect systems, algebraic models of sequential computation, and concrete effect systems.

9.1 Generic Effect Systems

We know of only three generic characterizations of effect systems prior to ours, none of which handles sequential effects or is extensible with new primitives.

Marino and Millstein give a generic model of a static commutative effect system [42] for a simple extension of the lambda calculus. Their formulation is motivated explicitly by the view of effects as capabilities, which pervades their formalism — effects there are sets of capabilities, values can be tagged with sets of capabilities, and subeffecting follows from set inclusion. They do not consider polymorphism (beyond the naive exponential-cost approach of substituting let bindings at type checking). They do however also parameterize their development by an insightful choice of *adjust* to change the capabilities available within some evaluation context and *check* to check the capabilities required by some redex against those available, allowing great flexibility in how effects are managed.

Henglein et al. [31] give a simple expository effect system to introduce the technical machinery added to a standard typing judgment in order to track (commutative) effects. Like like Marino and Millstein they use qualifiers as a primitive to introduce effects. Because their goals were instructional rather than technical, the calculus is not used for much (it precedes a full typed region calculus [55]).

Rytz et al. [50] offer a collection of insights for building manageable effect systems, notably the relative effect polymorphism mentioned earlier [49] (inspired by anchored exceptions [59]) and an approach for managing the simultaneous use of multiple effect systems with modest annotation burden. The system was given abstractly, with respect to a lattice of effects. Toro and Tanter later implemented this as as a polymorphic extension [58] to Schwerter et al.'s gradual effect systems [3]. Their implementation is again parameterized with respect to an effect lattice, supporting only closed effects (i.e., no singletons).

9.2 Algebraic Approaches to Computation

Our effect quantales are an example of an algebraic approach to modeling sequential computation. There are many closely-related approaches beyond those discussed in Section 7, such as action logic [48] and Kleene Algebras (KAs), and Kleene Algebras with Tests (KATs) [39]. Each of these has some partial order, and an associative binary operation that distributes over joins (and meets). Some KAs also look very much like effect quantales: one standard example is a KA of execution traces, similar to the effect systems mentioned in Section 4.3. However, Kleene Algebras and relatives are intended to model the semantics of a *possibly-failing* computation, rather than a classification of "successful" computations, and thus carries a ring structure unsuitable for effect systems. The requirement that the KA element 0 of the partial order is nilpotent for sequencing ($0 \cdot x = 0 = x \cdot 0$) but also least in the partial order ($0 + x = x = x + 0$) makes these systems unsuitable for effect systems. Some effects have no sensible least element: for locking, this would be an effect e that is considered to both preserve lock sets ($e \sqsubseteq (\varnothing, \varnothing)$) and also change them (e.g., $e \sqsubseteq (\varnothing, \{\ell\})$ among others). For those systems where a least element does make sense (atomicity without locking, or subsuming commutative effects), their least element $\bot$ is always the identity for sequencing — $\bot \rhd x = x = x \rhd \bot$. The ring requirements would require $A \rhd B \rhd A = B$ for atomicity, which fails to reflect that such a sequence is not atomic.

9.3 Concrete Effect Systems

We discussed several example sequential effect systems throughout, notably Flanagan and Abadi's *Types for Safe Locking* [18] (the precursor to RCC/Java [19]), and Flanagan and Qadeer's *Types for Atomicity* [21] (again a precursor to a full Java version [20]). This atomicity work is one of the best-known examples of a sequential effect system. Coupling the atomicity structures developed there with a sequential version of lockset tracking for unstructured locking primitives gives rise to interesting effect quantales, which can be separately specified and then combined to yield a complete effect system.

Suenaga gives a sequential effect system for ensuring deadlock freedom in a language with unstructured locking primitives [53], which is the closest example we know of to our lockset effect quantale. However, Suenaga's lock tracking is structured a bit differently from ours: he tracks the state of a lock as either explicitly present but unowned (by the current thread), or owned by the current thread, thus not reasoning about recursive lock acquisition. This is isomorphic to a *set*, rather than a multiset, of locks (a subset of a known set of all locks), and thus checks a different property than our lockset quantale. In fact, most prior type systems tracking owned locks treat only this binary property. This discrepancy between prior work and our lockset quantales leads to interesting, and slightly surprising subtleties.

Our first attempt to define the locking effect quantale sought to use only *sets* of locks, rather than *multisets*, and to prohibit recursive lock acquisition. Indeed, such an effect quantale can be defined, satisfying all required properties, for a fixed set of locks. But once the set of locks is a parameter, the resulting indexed effect quantale is not collapsible! Viewing this in terms of the type system, consider the term $f = (\lambda\, l_1 .\, \lambda\, l_2 .\, \mathsf{acquire}\ l_1;\ \mathsf{acquire}\ l_2)$, which would have type $\Pi l_1 : \mathsf{lock} \xrightarrow{I} \Pi l_2 : \mathsf{lock} \xrightarrow{(\varnothing,\{l_1,l_2\})} \mathsf{unit}$ (ignoring atomicity). Intuitively, applying this function to the same lock x twice ($f\ x\ x$) would eventually substitute the same value for l_1 and l_2, yielding an expected overall effect of $(\varnothing, \{x\})$ — the number of locks acquired shrank because the set would collapse, though the underlying term would try to acquire the same lock twice. Moreover, after reducing the second application, the resulting term would no longer by type-correct, as $(\varnothing, \{x\}) \rhd (\varnothing, \{x\}) = \mathsf{Err}$ when holding a lock twice cannot be represented! This is why the *set*-based lock tracking is not collapsible. Using multisets as we do in Section 4 fixes this problem. Suenaga does not encounter this, because his lack of closures and linear lock ownership do not permit two variables used for locking to later be unified by substitution. Other work such as RCC/Java [19] avoids the issue because while the system uses sets, the dynamic semantics permit recursive acquisition and count recursive claims in the evaluation contexts.

Many other systems that are not typically presented as effect systems can be modeled as sequential effect systems. Notably this includes systems with flow-sensitive additional contexts (e.g., sets of capabilities) as alluded to in Section 2, or fragments of type information in systems that as-presented perform strong updates on the local variable contexts (e.g., the state transitions tracked by typestate [60, 24], though richer systems require dynamic reflection of typestate checks into types [54], which is a richer form of dependent effects than our framework currently tracks). Other forms of behavioral type systems have at least a close correspondence to known effect systems, which are likely to be adaptable to our framework in the future: consider the similarity between session types [32] and Nielson and Nielson's effect system for communication in CML [46].

9.4 Limitations and Future Work

There remain a few important aspects of sequential effect systems that neither we, nor related work on semantic characterizations of sequential effects, have considered. One important example is the presence of a masking construct [41, 25] that locally suppresses some effect, such as try-catch blocks or `letregion` in region calculi. Another is serious consideration of control effects, which are alluded to in Mycroft et al.'s work [45], but otherwise have not been directly considered in the algebraic characterizations of sequential effects.

Our generic language carries some additional limitations. It lacks subtyping and "subeffecting," which enhance usability of the system, but these should not present any new technical difficulties. It also lacks support for adding new evaluation contexts through the parameters, which is important for modeling constructs like `letregion`. Allowing this would require more sophisticated machinery for composing partial semantic definitions [5, 12, 13].

Beyond the effect-flavored variation [41, 55] of parametric polymorphism and the polymorphism arising from singleton types as we consider here, the literature contains bounded [30] (or more generally, constraint-based) effect polymorphism, and unique "lightweight" forms of effect polymorphism [50, 27] with no direct parallel in traditional approaches to polymorphism. Extending our approach for these seems sensible and feasible.

Finally, we have not considered concurrency and sequential effects, beyond noting the gap between joinoids' fork-join style operator and common source-level concurrency constructs. As a result we have not directly proven that our multiset-of-locks effect quantale ensures data race freedom or atomicity for a true concurrent language.

10 Conclusions

We have given a new algebraic characterization — effect quantales — for sequential effect systems, and shown it sufficient to implement complete effect systems, unlike previous approaches that focused on a subset of real language features. We used them to model classic examples from the sequential effect system literature, and gave a syntactic soundness proof for the first generic sequential effect system. Moreover, we give the first investigation of the generic interaction between (singleton) dependent effects and algebraic models of sequential effects, and a powerful way to derive an appropriate iteration operator on effects for many effect quantales. We believe this is an important basis for future work designing complete sequential effect systems, and for generic effect system implementation frameworks supporting sequential effects.

References

1 Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, March 2006. doi:10.1145/1119479.1119480.

2 Samson Abramsky and Steven Vickers. Quantales, observational logic and process semantics. *Mathematical Structures in Computer Science*, 3(02):161–227, 1993. doi:10.1017/S0960129500000189.

3 Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 283–295. ACM, 2014. doi:10.1145/2628136.2628149.

4 Nick Benton and Peter Buchlovsky. Semantics of an Effect Analysis for Exceptions. In *TLDI*, 2007. doi:10.1145/1190315.1190320.

5 Lars Birkedal and Rasmus Ejlers Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*, pages 213–222. IEEE, 2013. doi:10.1109/LICS.2013.27.

6 Garrett Birkhoff. *Lattice theory*, volume 25 of *Colloquium Publications*. American Mathematical Soc., 1940. Third edition, eighth printing with corrections, 1995.

7 Thomas Scott Blyth. *Lattices and ordered algebraic structures*. Springer Science & Business Media, 2006. doi:10.1007/b139095.

8 Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009. doi:10.1145/1640089.1640097.

9 Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002. doi:10.1145/582419.582440.

10 Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2001. doi:10.1145/504282.504287.

11 Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–275. ACM, 1999. doi:10.1145/292540.292564.

12 Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 207–218. ACM, 2013. doi:10.1145/2429069.2429094.

13 Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. Modular monadic meta-theory. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 319–330. ACM, 2013. doi:10.1145/2500365.2500587.

14 Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *Proceedings of the 18th European Symposium on Programming (ESOP)*, pages 363–377. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-00590-9_26.

15 Peter Dybjer. Internal type theory. In *International Workshop on Types for Proofs and Programs*, pages 120–134. Springer, 1995. doi:10.1007/3-540-61780-9_66.

16 Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 177–190. ACM, 2006. doi:10.1145/1217935.1217953.

17 Cormac Flanagan and Martín Abadi. Object Types against Races. In *CONCUR*, 1999. doi:10.1007/3-540-48320-9_21.

18 Cormac Flanagan and Martín Abadi. Types for Safe Locking. In *ESOP*, 1999. doi:10.1007/3-540-49099-X_7.

19 Cormac Flanagan and Stephen N. Freund. Type-Based Race Detection for Java. In *PLDI*, 2000. doi:10.1145/349299.349328.

20 Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 338–349. ACM, 2003. doi:10.1145/781131.781169.

21 Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, pages 1–12. ACM, 2003. doi:10.1145/604174.604176.

22 Laszlo Fuchs. *Partially ordered algebraic systems*, volume 28 of *International Series of Monographs on Pure and Applied Mathematics*. Dover Publications, 2011. Reprint of 1963 Pergamon Press version.

23 Nikolaos Galatos, Peter Jipsen, Tomasz Kowalski, and Hiroakira Ono. *Residuated lattices: an algebraic glimpse at substructural logics*, volume 151 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2007.

24 Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12:1–12:44, October 2014. doi:10.1145/2629609.

25 David K. Gifford and John M. Lucassen. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, 1986. doi:10.1145/319838.319848.

26 Colin S. Gordon. A Generic Approach to Flow-Sensitive Polymorphic Effects (Extended Version). Technical Report arXiv cs.PL 1705.02264, Computing Research Repository (CoRR), May 2017. URL: https://arxiv.org/abs/1705.02264.

27 Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. JavaUI: Effects for Controlling UI Object Access. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*, 2013. doi:10.1007/978-3-642-39038-8_8.

28 Colin S. Gordon, Michael D. Ernst, and Dan Grossman. Static Lock Capabilities for Deadlock Freedom. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'12)*, 2012. doi:10.1145/2103786.2103796.

29 James Gosling, Bill Joy, Guy L Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification: Java SE 8 Edition*. Pearson Education, 2014.

30 Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293. ACM, 2002. doi:10.1145/512529.512563.

31 Fritz Henglein, Henning Makholm, and Henning Niss. Effect types and region-based memory management. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 3, pages 87–136. MIT Press, 2005.

32 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, 2008. doi:10.1145/1328438.1328472.

33 Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing os processes to improve dependability and safety. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 341–354. ACM, 2007. doi:10.1145/1272996.1273032.

34 Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007. doi:10.1145/1243418.1243424.

35 Bart Jacobs. *Categorical logic and type theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1999.

36 Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 633–645. ACM, 2014. doi:10.1145/2535838.2535846.

37 Ming Kawaguchi, Patrick Rondon, Alexander Bakst, and Ranjit Jhala. Deterministic Parallelism via Liquid Effects. In *PLDI*, 2012. doi:10.1145/2254064.2254071.

38 Eric Koskinen and Tachio Terauchi. Local temporal reasoning. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 59:1–59:10, New York, NY, USA, 2014. ACM. doi:10.1145/2603088.2603138.

39 Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(3):427–443, 1997. doi:10.1145/256167.256195.

40 Richard J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications of the ACM*, 18(12):717–721, December 1975. doi:10.1145/361227.361234.

41 J. M. Lucassen and D. K. Gifford. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1988. doi:10.1145/73560.73564.

42 Daniel Marino and Todd Millstein. A Generic Type-and-Effect System. In *TLDI*, 2009. doi:10.1145/1481861.1481868.

43 Christopher J. Mulvey. &. *Suppl. Rend. Circ. Mat. Palermo (2)*, 12:99–104, 1986.

44 Christopher J Mulvey and Joan W Pelletier. A quantisation of the calculus of relations. In *Canad. Math. Soc. Conf. Proc. 13*, pages 345–360, 1992.

45 Alan Mycroft, Dominic Orchard, and Tomas Petricek. Effect systems revisited—control-flow algebra and semantics. In *Semantics, Logics, and Calculi*, pages 1–32. Springer, 2016. doi:10.1007/978-3-319-27810-0_1.

46 Flemming Nielson and Hanne Riis Nielson. From cml to process algebras. In *International Conference on Concurrency Theory (CONCUR)*, pages 493–508. Springer, 1993. doi:10.1007/3-540-57208-2_34.

47 Wesley Phoa. An introduction to fibrations, topos theory, the effective topos and modest sets. Technical Report ECS-LFCS-92-208, University of Edinburgh, 1992.

48 Vaughan Pratt. Action logic and pure induction. In *European Workshop on Logics in Artificial Intelligence*, pages 97–120. Springer, 1990. doi:10.1007/BFb0018436.

49 Lukas Rytz and Martin Odersky. Relative Effect Declarations for Lightweight Effect-Polymorphism. Technical Report EPFL-REPORT-175546, EPFL, 2012.

50 Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight Polymorphic Effects. In *European Conference on Object-Oriented Programming (ECOOP 2012)*, 2012. doi:10.1007/978-3-642-31057-7_13.

51 Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. The Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 333–352. ACM, 1991. doi:10.1145/99583.99627.

52 Christian Skalka. Types and trace effects for object orientation. *Higher-Order and Symbolic Computation*, 21(3):239–282, 2008. doi:10.1007/s10990-008-9032-6.

53 Kohei Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *Asian Symposium on Programming Languages and Systems*, pages 155–170. Springer, 2008. doi:10.1007/978-3-540-89330-1_12.

54 Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in plaid. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 713–732, New York, NY, USA, 2011. ACM. doi:10.1145/2048066.2048122.

55 Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of functional programming*, 2(03):245–271, 1992. doi:10.1017/S0956796800000393.

56 Ross Tate. The sequential semantics of producer effect systems. In *POPL '13: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. ACM, 2013. doi:10.1145/2429069.2429074.

57 Mads Tofte and Jean-Pierre Talpin. Implementation of the Typed Call-by-value λ-calculus Using a Stack of Regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 188–201, 1994. doi:10.1145/174675.177855.

58 Matías Toro and Éric Tanter. Customizable gradual polymorphic effects for scala. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 935–953. ACM, 2015. doi:10.1145/2814270.2814315.

59 Marko van Dooren and Eric Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 455–471. ACM, 2005. doi:10.1145/1094811.1094847.

60 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, pages 459–483, 2011. doi:10.1007/978-3-642-22655-7_22.

61 David N Yetter. Quantales and (noncommutative) linear logic. *The Journal of Symbolic Logic*, 55(01):41–64, 1990. doi:10.2307/2274953.